Armazenamento distribuído de dados e checkpointing de aplicações paralelas em grades oportunistas

Raphael Yokoiingawa de Camargo

TESE APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO DE DOUTOR
EM
CIÊNCIAS

Área de Concentração: Ciência da Computação
Orientador: Prof. Dr. Fabio Kon

Durante a elaboração deste trabalho o autor recebeu o auxílio financeiro do CNPq

São Paulo, 30 de Agosto de 2007

Armazenamento distribuído de dados e checkpointing de aplicações paralelas em grades oportunistas

Este exemplar corresponde à redação
final da tese devidamente corrigida
e defendida por Raphael Yokoingawa de Camargo
e aprovada pela Comissão Julgadora.

São Paulo, 30 de agosto de 2007

Banca Examinadora:

- Prof. Dr. Fabio Kon (orientador) - IME-USP.
- Prof. Dr. Marco Dimas Gubitoso - IME-USP
- Prof. Dr. Luiz Eduardo Buzato IC - Unicamp
- Prof. Dr. Markus Endler - PUC-Rio
- Prof. Dr. Francisco Brasileiro - UFCG

Dedico este trabalho à Selma, meu grande amor,
cuja compreensão, dedicação e carinho trouxeram equilíbrio e grandes alegrias à minha vida.

Agradecimentos

Agradeço de modo especial ao professor Fabio Kon que, com sua competente orientação, constantemente me motivou a superar os desafios da carreira acadêmica. Ao CNPq, pelo seu apoio financeiro durante todos estes anos.

A todos meus amigos do LCPD, das listas aabalados e psycohunters e da ETFSP. Dos simples almoços no bandeirão às longas baladas noturnas, sua companhia foi essencial para tornar minha vida mais alegre. E um agradecimento especial àqueles que ainda contribuíram com meu trabalho de doutorado.

Aos meus pais, por todo o apoio e carinho que possibilitaram o meu amadurecimento e a conclusão deste doutorado, e às minhas irmãs e meus familiares, por sua constante presença a meu lado.

À Selma, por me apoiar nos momentos mais difíceis. Seu afeto e compreensão me deram forças para superar este grande desafio que é o doutorado.

RESUMO

Grades computacionais oportunistas utilizam recursos ociosos de máquinas compartilhadas para executar aplicações que necessitam de um alto poder computacional e/ou trabalham com grandes quantidades de dados. Mas a execução de aplicações paralelas computacionalmente intensivas em ambientes dinâmicos e heterogêneos, como grades computacionais oportunistas, é uma tarefa difícil. Máquinas podem falhar, ficar inacessíveis ou passar de ociosas para ocupadas inesperadamente, comprometendo a execução de aplicações. Um mecanismo de tolerância a falhas que dê suporte a arquiteturas heterogêneas é um importante requisito para estes sistemas. Neste trabalho, analisamos, implementamos e avaliamos um mecanismo de tolerância a falhas baseado em checkpointing para aplicações paralelas em grades computacionais oportunistas. Este mecanismo permite o monitoramento de execuções e a migração de aplicações entre nós heterogêneos da grade.

Mas além da execução, é preciso gerenciar e armazenar os dados gerados e utilizados por estas aplicações. Desejamos uma infra-estrutura de armazenamento de dados de baixo custo e que utilize o espaço livre em disco de máquinas compartilhadas da grade. Devemos utilizar somente os ciclos ociosos destas máquinas para armazenar e recuperar dados, de modo que um sistema de armazenamento distribuído que as utilize deve ser redundante e tolerante a falhas.

Para resolver o problema do armazenamento de dados em grades oportunistas, projetamos, implementamos e avaliamos o middleware OppStore. Este middleware provê armazenamento distribuído e confiável de dados, que podem ser acessados de qualquer máquina da grade. As máquinas são organizadas em aglomerados, que são conectados por uma rede *peer-to-peer* auto-organizável e tolerante a falhas. Dados são codificados em fragmentos redundantes antes de serem armazenados, de modo que arquivos podem ser reconstruídos utilizando apenas um subconjunto destes fragmentos. Finalmente, para lidar com a heterogeneidade dos recursos, desenvolvemos uma extensão ao protocolo de roteamento em redes *peer-to-peer* Pastry. Esta extensão adiciona balanceamento de carga e suporte à heterogeneidade de máquinas ao protocolo Pastry.

ABSTRACT

Opportunistic computational grids use idle resources from shared machines to execute applications that need large amounts of computational power and/or deal with large amounts of data. But executing computationally intensive parallel applications in dynamic and heterogeneous environments, such as opportunistic grids, is a daunting task. Machines may fail, become inaccessible, or change from idle to occupied unexpectedly, compromising the application execution. A fault tolerance mechanism that supports heterogeneous architectures is an important requisite for such systems. In this work, we analyze, implement and evaluate a checkpointing-based fault tolerance mechanism for parallel applications running on opportunistic grids. The mechanism monitors application execution and allows the migration of applications between heterogeneous nodes of the grid.

But besides application execution, it is necessary to manage data generated and used by those applications. We want a low cost data storage infrastructure that utilizes the unused disk space of grid shared machines. The system should use the machines to store and recover data only during their idle periods, requiring the system to be redundant and fault-tolerant.

To solve the data storage problem in opportunistic grids, we designed, implemented and evaluated the OppStore middleware. This middleware provides reliable distributed storage for application data, which can be accessed from any machine in the grid. The machines are organized in clusters, connected by a self-organizing and fault-tolerant peer-to-peer network. During storage, data is codified into redundant fragments, allowing the reconstruction of the original file using only a subset of those fragments. Finally, to deal with resource heterogeneity, we developed an extension to the Pastry peer-to-peer routing substrate, enabling heterogeneity-aware load-balancing message routing.

Sumário

1	Introdução	1
1.1	Motivação	4
1.2	Objetivos	5
1.3	Organização da Tese	5
2	Checkpointing e Tolerância a Falhas	7
2.1	Checkpointing de aplicações paralelas	8
2.1.1	Conceitos e terminologia	9
2.1.2	Protocolos de <i>checkpointing</i>	11
2.2	Abordagens para <i>checkpointing</i> de processos	14
2.2.1	Checkpointing no nível do sistema	14
2.2.2	Checkpointing no nível da aplicação	15
2.2.3	Obtenção do estado de um processo no nível da aplicação	16
2.3	Execução de aplicações paralelas em arquiteturas heterogêneas	18
2.3.1	O modelo de programação BSP	18
2.3.2	Comunicação portátil entre processos	21
2.4	Armazenamento de checkpoints	21
2.4.1	Estratégias de armazenamento	22
2.5	Resumo	25
3	Tolerância a Falhas de Aplicações no Sistema InteGrade	27
3.1	Trabalhos relacionados	27
3.1.1	Checkpointing no nível da aplicação	28
3.1.2	Checkpointing de aplicações paralelas	29
3.1.3	Checkpointing em grades computacionais	29
3.1.4	Estratégias de armazenamento	30
3.2	Visão geral de recuperação por retrocesso no InteGrade	31
3.2.1	Arquitetura do InteGrade	31

3.2.2	Recuperação por retrocesso baseada em <i>checkpointing</i>	33
3.3	Implementação dos principais componentes	35
3.3.1	Módulos de Gerenciamento	35
3.3.2	Componentes para provedores de recursos	39
3.3.3	Reinicialização de processos e aplicações	39
3.4	Pré-compilador e biblioteca de <i>checkpointing</i>	42
3.4.1	Pré-compilador	42
3.4.2	Instrumentação do código de uma aplicação	43
3.4.3	Criação de <i>checkpoints</i> e recuperação do estado da aplicação	49
3.4.4	Transformação de dados	51
3.4.5	Armazenamento e recuperação de checkpoints	52
3.5	Aplicações paralelas BSP	53
3.5.1	Execução de aplicações BSP em máquinas de diferentes arquiteturas	53
3.5.2	<i>Checkpointing</i> de aplicações BSP	54
3.6	Resumo	54
4	Avaliação do Mecanismo de <i>Checkpointing</i>	55
4.1	Sobrecarga causada pelo mecanismo de <i>checkpointing</i>	56
4.2	Execução de uma aplicação BSP na presença de falhas	58
4.3	Execução de uma aplicação BSP em nós heterogêneos	59
4.4	Recuperação do estado contido em um checkpoint	60
4.5	Estratégias de armazenamento	61
4.5.1	Codificação e decodificação de dados	61
4.5.2	Sobrecarga sobre o tempo de execução	62
4.6	Resumo	64
5	Armazenamento Distribuído de Dados	67
5.1	Sistemas <i>peer-to-peer</i>	68
5.1.1	Redes <i>peer-to-peer</i> não-estruturadas	69
5.1.2	Redes <i>peer-to-peer</i> estruturadas	70
5.1.3	Balanceamento de carga em DHTs	72
5.2	Armazenamento distribuído de dados em redes <i>peer-to-peer</i>	74
5.2.1	CFS	74
5.2.2	PAST	75
5.2.3	OceanStore	75
5.2.4	Outros sistemas de armazenamento <i>peer-to-peer</i>	76
5.3	Armazenamento de dados em grades computacionais	76

5.3.1	Grades de dados	76
5.3.2	Transporte de dados	77
5.3.3	Armazenamento e gerenciamento de réplicas	78
5.3.4	Armazenamento com redes <i>peer-to-peer</i>	79
5.4	Resumo	80
6	Identificadores Virtuais	81
6.1	Trabalhos relacionados	82
6.2	Pastry	83
6.2.1	Protocolos	84
6.3	Identificadores virtuais	85
6.3.1	Visão geral dos protocolos e tabelas	86
6.4	Protocolos sobre Pastry	88
6.4.1	Protocolo de roteamento virtual	88
6.4.2	Protocolo de partição do espaço virtual	89
6.4.3	Protocolo de ingresso de nós	91
6.4.4	Protocolo de saída de nós	92
6.4.5	Protocolo de atualização de capacidade	92
6.5	Implementação sobre o FreePastry	93
6.5.1	FreePastry	93
6.5.2	Arquitetura	94
6.5.3	Conjunto de folhas e vizinhos virtuais	95
6.5.4	Protocolos	96
6.6	Resumo	99
7	OppStore: Middleware para Armazenamento Distribuído de Dados	101
7.1	Trabalhos Relacionados	101
7.2	Projeto do middleware OppStore	103
7.2.1	Armazenamento e recuperação de dados	105
7.2.2	Identificadores virtuais no OppStore	108
7.3	Principais componentes	109
7.3.1	Gerenciador de repositórios de dados do aglomerado (CDRM)	109
7.3.2	Repositório autônomo de dados (ADR)	110
7.3.3	Intermediador de acesso (<i>Access Broker</i>)	111
7.4	Gerenciamento de dados armazenados	112
7.4.1	Disponibilidade de dados	112
7.4.2	Gerenciamento de arquivos	113

7.4.3	Gerenciamento de fragmentos	114
7.4.4	Otimização de desempenho	115
7.5	Implantação sobre grades computacionais	115
7.5.1	Interface com grades computacionais	115
7.5.2	InteGrade	116
7.5.3	Condor	117
7.6	Implementação	118
7.6.1	Principais componentes	118
7.6.2	Gerenciamento de dados	122
7.6.3	Interface com o InteGrade	124
7.7	Resumo	125
8	Avaliação do middleware OppStore	127
8.1	Simulações	127
8.1.1	Ambiente de simulação	127
8.1.2	Avaliação do uso de identificadores virtuais	129
8.1.3	Recuperação de dados armazenados no OppStore	133
8.2	Experimentos	135
8.2.1	Armazenamento de dados	136
8.2.2	Recuperação de dados	137
8.3	Resumo	137
9	Conclusões	139
9.1	Trabalhos Futuros	140
9.2	Contribuições do Trabalho	141
9.2.1	Contribuições tecnológicas	141
9.2.2	Contribuições científicas	142
9.3	Publicações durante o doutorado	143

Capítulo 1

Introdução

Existe uma grande classe de aplicações que necessitam de um alto poder computacional e trabalham com grandes quantidades de dados. Estas aplicações incluem o seqüenciamento de genes [115, 133], enovelamento de proteínas [93, 134], análise de sinais (ex: SETI [7]), análises financeiras [79], mineração de dados [47], física de partículas [13] e simulações em engenharia. Mas cientistas, pesquisadores, analistas e engenheiros muitas vezes não têm acesso a uma infra-estrutura computacional que lhes permita a execução destas aplicações, normalmente por trabalharem em instituições que dispõem de recursos limitados. Por outro lado, estas mesmas instituições normalmente possuem centenas ou milhares de máquinas utilizadas pelos seus membros e que permanecem ociosas pela grande maioria do tempo. Se pudéssemos utilizar os ciclos computacionais ociosos e o espaço livre em disco destas máquinas, seria possível realizar a execução de uma parte significativa das aplicações acima citadas.

Condor [81] foi um dos primeiros sistemas desenvolvidos especificamente para utilizar os ciclos ociosos de estações de trabalho compartilhadas. Condor mantém uma fila de requisições, que são escalonadas para execução em máquinas de uma rede local que estiverem ociosas. Mas quando o dono de uma destas máquinas ociosas resolve utilizá-la novamente, este deve recuperar o controle sobre todos seus recursos computacionais. Os processos pertencentes ao Condor sendo executados nesta máquina são então finalizados e migrados para outras máquinas ociosas ou entram novamente na fila de requisições.

O termo Computação em Grade (*Grid computing*) [16, 38, 49, 113] foi definido para denominar uma infra-estrutura que permite o compartilhamento de recursos heterogêneos e dispersos geograficamente para a execução de aplicações. Estas infra-estruturas normalmente oferecem serviços de descoberta e compartilhamento de recursos, segurança, escalonamento, tolerância a falhas, entre outros. Como comparação, um *cluster* normalmente consiste de um grupo de computadores homogêneos conectados em uma rede local onde usuários podem executar aplicações.

Uma classe de grades computacionais, denominada grades oportunistas [33, 55, 131], utiliza estações de trabalho compartilhadas para executar as aplicações da Grade. O objetivo é utilizar o poder computacional ocioso destas máquinas para realizar computação útil, de modo a aumentar o poder com-

putacional de uma instituição sem a necessidade de adquirir hardware adicional. O InteGrade¹ [55] é um middleware que permite a criação de grades computacionais oportunistas. Ele é organizado como uma federação de aglomerados², onde cada aglomerado contém máquinas que disponibilizam seus recursos ociosos para utilização por aplicações da grade. Versões mais recentes do Condor [34] possuem suporte para a instanciamento de múltiplos aglomerados e também se encaixam nesta categoria de grade computacional oportunista.

Para possibilitar o uso destes recursos ociosos, necessitamos de sistemas de middleware específicos para grades oportunistas. Mas o desenvolvimento destes sistemas requer que tratemos de diversos desafios: (1) segurança, tanto para os usuários compartilhando seus recursos como para as aplicações e dados que utilizam estes recursos, (2) escalonamento de aplicações, (3) tolerância a falhas, tanto dos componentes da grade quanto para as aplicações sendo executadas, (4) armazenamento e gerenciamento de dados de usuários e aplicações e (5) manutenção da qualidade de serviço percebida pelos donos das máquinas compartilhadas. No projeto de pesquisa InteGrade, alunos de mestrado e doutorado de diferentes instituições paralelamente realizam pesquisas para tentar solucionar cada um destes problemas. Neste trabalho de doutoramento, tratamos de dois problemas importantes: a execução robusta de aplicações paralelas em máquinas não-dedicadas e o armazenamento de dados de usuários e aplicações da grade.

Garantir a *execução robusta de aplicações paralelas em máquinas não-dedicadas* pertencentes a um ambiente dinâmico e heterogêneo, como o de grades oportunistas, é uma tarefa difícil. Máquinas podem falhar, ficar indisponíveis ou mudar de ociosas para ocupadas inesperadamente, comprometendo a execução das aplicações. Diferentemente de máquinas dedicadas, cujo tempo médio entre falhas é tipicamente da ordem de semanas [83], máquinas não-dedicadas podem se tornar indisponíveis diversas vezes em um único dia. Um mecanismo de tolerância a falhas, como a recuperação por retrocesso baseada em *checkpointing* [46], pode ser utilizado para garantir a progressão da execução de uma aplicação mesmo na presença de falhas. Para tal, *checkpoints* contendo o estado de uma aplicação são periodicamente gerados e armazenados, permitindo a reinicialização da aplicação, em caso de falha em um de seus processos, a partir de um estado intermediário de sua execução.

Mas a grande maioria dos mecanismos de *checkpointing* requerem que um processo seja reiniciada em uma máquina homogênea com relação àquela onde o *checkpoint* foi gerado (normalmente deve possuir pelo menos o mesmo hardware e versão do sistema operacional). Como as grades oportunistas são tipicamente compostas por máquinas heterogêneas pertencentes a diferentes instituições, para permitir que *checkpoints* de processos gerados em uma máquina possam ser recuperados em outra com diferentes características, é necessário que o mecanismo de *checkpointing* armazene os estados das apli-

¹<http://www.integrate.org.br>

²Ao tratarmos de grades oportunistas, definimos um aglomerado como sendo um conjunto de máquinas pertencentes a um único domínio administrativo, como um laboratório ou departamento de uma instituição. Estes aglomerados da grade podem incluir desde *clusters* dedicados até máquinas pessoais compartilhadas.

cações de modo portátil. Além disso, no caso de aplicações paralelas, o mecanismo precisa considerar as dependências entre os processos da aplicação ao obter seu estado global.

Finalmente, *checkpoints* gerados precisam ser salvos em um meio de armazenamento estável. Normalmente a execução de um processo falha porque a máquina onde este era executado ficou indisponível, de modo que o disco desta máquina não pode ser considerado como um meio estável. A solução mais comum é instalar servidores dedicados para o armazenamento remoto de checkpoints. Mas esta abordagem tem algumas desvantagens, como falta de escalabilidade e necessidade de servidores dedicados. Uma solução para o problema seria distribuir os dados em outros nós do aglomerado. Mas como estas máquinas são não-dedicadas, técnicas de replicação e codificação de dados precisam ser aplicadas para garantir a disponibilidade dos *checkpoints* armazenados, o que nos leva ao segundo problema abordado na tese.

O segundo problema que tratamos é o de *armazenamento distribuído de dados de aplicações da grade*. Além de ciclos computacionais, aplicações para grades tipicamente requerem ou produzem grandes quantidades de dados. Dados de saída de uma aplicação podem ser utilizados por diversas outras, criando a necessidade de um espaço de armazenamento compartilhado. Uma infra-estrutura de armazenamento de dados de uma grade deve ser portanto confiável, de alta capacidade e acessível de qualquer ponto da grade.

Tradicionalmente, grades computacionais utilizam servidores de dados dedicados para o armazenamento e compartilhamento de dados. Estes servidores mantêm diversas réplicas dos dados armazenados, de modo que sistemas de gerenciamento de réplicas [24, 30, 106] são comuns nestes cenários. Grades de dados [31, 138] são projetadas para permitir a execução de aplicações científicas que requerem grandes quantidades de dados, da ordem de Terabytes ou até Petabytes, rodam em supercomputadores e necessitam de redes especializadas de alta-velocidade. Estas grades são direcionadas a grupos de pesquisa de instituições que possuem abundância de recursos financeiros, onde o custo de implantação de uma grade, apesar de importante, não é um fator decisivo.

Neste trabalho, desejamos criar uma infra-estrutura de armazenamento de dados direcionada a uma classe de aplicações que podem ser executadas em grades computacionais oportunistas. Estas grades são compostas por máquinas compartilhadas, que tipicamente possuem quantidades significativas de espaço livre em disco. Ao combinarmos o espaço livre em disco de algumas dezenas ou centenas de máquinas, podemos facilmente obter Terabytes de espaço de armazenamento. Utilizar estes recursos ociosos permitiria que obtivéssemos grandes quantidades de espaço de armazenamento a um baixo custo e sem a aquisição de hardware extra.

Mas utilizar máquinas compartilhadas para o armazenamento de dados não é uma tarefa simples. É preciso preservar a qualidade de serviço de donos de recursos computacionais compartilhados, de forma que devemos utilizar estes recursos somente durante seus períodos de ociosidade. Além disso, estas má-

quinas são, em alguns casos, desligadas, por exemplo, no período noturno ou fins de semana. Um sistema de armazenamento distribuído que utilize estas máquinas deve garantir a disponibilidade dos dados neste ambiente altamente dinâmico, utilizando, por exemplo, replicação e/ou codificação dos dados armazenados. Finalmente, grades computacionais podem conter centenas de milhares de máquinas, requerendo que o sistema seja auto-organizável e altamente escalável.

Em uma infra-estrutura de armazenamento, é necessário determinar onde os dados serão armazenados e, posteriormente, localizá-los quando estes forem requisitados. O mecanismo de localização deve ser descentralizado, de modo a permitir uma maior escalabilidade do sistema. Um cenário similar, onde o espaço em disco livre de muitas máquinas é utilizado para armazenar dados de modo distribuído, é explorado por sistemas de arquivos distribuídos baseados em redes *peer-to-peer* [35, 108]. Estas iniciativas recentes apresentaram bons resultados, indicando que as redes sobrepostas do tipo *peer-to-peer* podem ser uma boa escolha para o armazenamento e localização de dados em grades oportunistas.

1.1 Motivação

Este trabalho foi motivado pelo desejo de permitir que recursos ociosos de estações de trabalho possam ser efetivamente utilizados para a execução de aplicações paralelas e o armazenamento distribuído de dados. Como membros do projeto InteGrade, esperamos permitir que universidades e instituições que não dispõem de recursos financeiros para adquirir novos equipamentos, freqüentemente caros, possam ampliar seu poder computacional disponível a um custo pequeno. Isto é particularmente importante para instituições onde a quantidade de recursos financeiros é escassa, situação comum em países em desenvolvimento.

Para a execução de aplicações paralelas em recursos compartilhados de grades oportunistas, mecanismos de *checkpointing* e recuperação por retrocesso são de grande importância. Apesar de *checkpointing* ser um conceito antigo, a maioria das pesquisas nesta área presumem sistemas homogêneos e a presença de repositórios dedicados para armazenar os checkpoints. Mas estas suposições não fazem sentido no contexto das grades oportunistas, onde as máquinas são heterogêneas e repositórios dedicados de *checkpoints* podem não existir. Desejamos desenvolver mecanismos de *checkpointing* e recuperação por retrocesso que atendam a estes novos requisitos.

A segunda motivação é avaliar a possibilidade de utilizar o espaço livre em disco das máquinas compartilhadas de uma grade oportunista. Com o aumento da capacidade dos discos rígidos, freqüentemente temos centenas de máquinas com grande quantidade de espaço livre em disco. A partir do desenvolvimento de um middleware, esperamos tornar este espaço em disco acessível aos usuários de grades oportunistas sem prejudicar os donos das máquinas que disponibilizam seus recursos.

1.2 Objetivos

O objetivo deste trabalho é desenvolver uma infra-estrutura de software que permita a utilização de máquinas não-dedicadas tanto para a execução de aplicações paralelas de longa duração como para o armazenamento de dados. Para permitir a execução robusta de aplicações paralelas em máquinas não-dedicadas, o objetivo é desenvolver uma infra-estrutura de recuperação por retrocesso que utilize *checkpoints* portáteis. Para gerar *checkpoints* portáteis, utilizaremos a técnica de *checkpointing* no nível da aplicação, onde a própria aplicação é responsável por determinar quais dados devem ser salvos. Com a ajuda de um pré-compilador, podemos automatizar o processo de instrumentação da aplicação. Para o armazenamento dos *checkpoints* gerados, avaliamos maneiras de utilizar outras máquinas do aglomerado onde a aplicação paralela é executada. Como estas máquinas não são dedicadas, é necessário utilizar técnicas de replicação e codificação para o armazenamento dos dados. Neste trabalho trataremos de três classes de aplicações: sequenciais, paralelas do tipo paramétrica e paralelas do tipo BSP [136].

Com relação ao armazenamento de dados de usuários e aplicações da grade, o objetivo é desenvolver um sistema de armazenamento distribuído que utilize os próprios nós da grade para realizar o armazenamento de dados. Um sistema deste tipo requer uma infra-estrutura sofisticada, incluindo mecanismos de busca, de tolerância a falhas e de segurança. Para atender a estes requisitos, desenvolvemos um sistema que utiliza uma rede *peer-to-peer* para a busca e armazenamento de dados. Os sistemas *peer-to-peer* tradicionais possuem um suporte insuficiente para heterogeneidade e balanceamento de carga, de modo que precisamos desenvolver uma extensão para um destes protocolos. Além disso, para prover tolerância a falhas ao sistema, utilizaremos técnicas de replicação e codificação de dados.

1.3 Organização da Tese

Este trabalho está dividido em duas partes. Na primeira, composta pelos Capítulos 2, 3 e 4, exploramos a utilização do mecanismo de *checkpointing* para permitir a execução de aplicações de longa duração. Na segunda parte, composta pelos Capítulos 5, 6, 7 e 8, avaliamos o armazenamento distribuído de dados em grades oportunistas. Finalmente, no Capítulo 9, apresentamos as conclusões deste trabalho e perspectivas de trabalhos futuros.

Mais especificamente, no Capítulo 2 apresentamos conceitos relacionados a *checkpointing* de aplicações paralelas e portabilidade de *checkpoints* gerados. No Capítulo 3 apresentamos o middleware InteGrade, a infra-estrutura necessária para realizar a recuperação por retrocesso baseada em *checkpointing* e a implementação do mecanismo de *checkpointing*. Experimentos que avaliam o mecanismo de *checkpointing* são apresentados no Capítulo 4. No Capítulo 5, discutimos o armazenamento distribuído de dados e introduzimos as redes *peer-to-peer*. No Capítulo 6, apresentamos o conceito de identificadores virtuais para balanceamento de carga nestas redes, enquanto no Capítulo 7 apresentamos o projeto e implementação do middleware OppStore. Resultados experimentais e de simulações para a avaliação do

OppStore são apresentados no Capítulo 8.

Capítulo 2

Checkpointing e Tolerância a Falhas

Num ambiente de grades oportunistas, onde recursos podem ficar indisponíveis a qualquer momento, um mecanismo de tolerância a falhas é essencial. No caso de aplicações paralelas que utilizam dezenas de máquinas e cuja execução se estende por muitas horas, a falha de uma única máquina neste período normalmente faz com que toda a computação já realizada seja perdida. Deste modo, numa grade oportunista, onde máquinas ficam indisponíveis várias vezes num único dia, a execução deste tipo de aplicação sem um mecanismo de tolerância a falhas é inviável.

A *recuperação por retrocesso baseada em checkpointing* [46] é um mecanismo que permite reiniciar uma execução interrompida de uma aplicação a partir de um *checkpoint* gerado anteriormente. Um *checkpoint*, ou ponto de salvaguarda, contém o estado consistente de uma aplicação em um determinado ponto de sua execução. Um estado é considerado consistente se ele é um estado possível de ser atingido pela aplicação durante alguma execução. Denominamos *checkpointing* [97] o processo de gerar um *checkpoint* contendo o estado de uma aplicação. Existem duas abordagens distintas para realizar o *checkpointing* de uma aplicação, no nível de sistema e no nível da aplicação, que diferem no modo como é realizada a captura do estado da aplicação.

Quando tratamos de aplicações paralelas acopladas, isto é, aplicações onde existem trocas de mensagens entre os processos, é necessário um cuidado extra de modo a garantir que os estados gerados sejam consistentes. Os processos desta aplicação paralela podem possuir dependências entre si, de modo que se salvarmos o estado de cada processo isoladamente, o estado global da aplicação pode não ser um estado válido. Diversos protocolos para *checkpointing* de aplicações paralelas foram propostos para garantir a geração de *checkpoints* consistentes [5, 10, 29, 46, 53, 80, 88].

Mas protocolos e técnicas para gerar *checkpoints* são apenas uma parte do processo de recuperação por retrocesso. É necessário ainda determinar onde e como armazenar estes *checkpoints*, bem como definir algoritmos para encontrar e eliminar *checkpoints* inúteis. Além disso, é preciso um mecanismo para detectar falhas de aplicações e reiniciá-las a partir do último *checkpoint* salvo.

Para realizar a recuperação por retrocesso poderíamos, além de gerar *checkpoints*, fazer um registro

dos eventos não-determinísticos ocorridos durante a execução da aplicação. Estes eventos normalmente correspondem a interações com o usuário e outros recursos externos, como impressoras. Utilizando estes registros é possível fazer com que a aplicação retorne deterministicamente a um estado posterior ao armazenado no *checkpoint* utilizado. Este processo é denominado *recuperação por retrocesso baseada em registros* [6]. Uma vez que em grades computacionais existe pouca interação com o usuário e recursos externos e, dada a complexidade de se implementar um sistema de registro de eventos, decidimos por não utilizar esta abordagem.

Neste capítulo apresentaremos os principais conceitos relacionados ao processo de *checkpointing*, enquanto no Capítulo 3 discutiremos a infra-estrutura que implementamos para realizar a recuperação por retrocesso no contexto de uma grade computacional oportunista. Iniciaremos este capítulo com a Seção 2.1, onde analisamos as principais dificuldades e os protocolos para realizar *checkpointing* de aplicações paralelas. Na Seção 2.2 discutimos as duas principais abordagens para *checkpointing* de processos. Na Seção 2.3 apresentamos o modelo de programação BSP e como executar aplicações paralelas em máquinas de arquiteturas heterogêneas. Na Seção 2.4, avaliamos as diferentes estratégias para armazenamento de *checkpoints* em um aglomerado de computadores. Finalizamos o capítulo com um resumo e discussão do que foi apresentado.

2.1 Checkpointing de aplicações paralelas

Quando tratamos de aplicações paralelas acopladas, é necessário considerar as dependências entre os processos da aplicação. Numa aplicação paralela, são gerados *checkpoints globais* constituídos pelos *checkpoints* produzidos por cada processo da aplicação. Mas estes *checkpoints* globais não formam necessariamente um estado global consistente. Este problema ocorre devido a uma possível dependência de causalidade entre os processos, produzida devido à troca de mensagens. Deste modo, quando os *checkpoints* dos processos são gerados de maneira arbitrária, é possível que, após uma falha, os processos da aplicação paralela tenham que utilizar *checkpoints* anteriores ao último. No pior caso, a aplicação pode ser forçada a reiniciar a partir de seu estado inicial. Este problema não ocorre com aplicações não-acopladas do tipo *bag-of-tasks*, onde a execução de cada processo é independente dos demais.

Diferentes protocolos de *checkpointing* tratam este problema de maneiras distintas. Alguns deixam os processos da aplicação escolherem livremente seus *checkpoints* enquanto outros requerem uma coordenação global deste processo de escolha, de modo a garantir que todos os *checkpoints* gerados formem um estado global consistente.

Iniciamos esta seção apresentando os conceitos e a terminologia necessários para definirmos os protocolos de *checkpointing* para aplicações paralelas. Em seguida, discutiremos as principais classes de protocolos existentes. Nesta seção consideraremos um modelo de aplicação paralela genérico e, no Capítulo 3, trataremos especificamente dos modelos de aplicação aos quais o InteGrade fornece suporte.

2.1.1 Conceitos e terminologia

Nesta seção apresentamos os principais conceitos e a terminologia relacionados à área de *checkpointing* de aplicações paralelas, como precedências causais, estados globais consistentes e linhas de recuperação.

Modelo de aplicação paralela

Consideramos um modelo de aplicação paralela composta por um conjunto de n processos que se comunicam exclusivamente através de trocas de mensagens assíncronas. Estas mensagens são enviadas por canais de comunicação confiáveis do tipo FIFO (*First In, First Out*), isto é, com garantia de entrega de mensagens na ordem em que foram enviadas. Os processos da aplicação são independentes, não possuindo acesso a um relógio global.

A execução de um processo é definida como uma seqüência de eventos $(e_i^0, e_i^1, \dots)$, onde e_i^γ representa o γ -ésimo evento associado ao processo i . Existem dois tipos de eventos, internos e de comunicação. Os eventos internos são associados às mudanças de estado do processo e os de comunicação correspondem ao envio e entrega de mensagens.

Precedências causais

O estado global de um sistema é caracterizado pelo conjunto dos estados de todos os processos participantes mais o estado dos canais de comunicação. Este estado global é consistente se, e somente se, é um estado que poderia ocorrer durante uma execução normal dos processos.

No modelo de sistema que estamos considerando, o fator causador de inconsistências são as trocas de mensagens entre processos. Por exemplo, um estado onde um processo recebeu uma mensagem que ainda não foi enviada é considerado inconsistente. Neste caso, temos uma relação de dependência entre os eventos de envio e recebimento da mensagem que não foi respeitada.

Para entender melhor essas dependências entre eventos, utilizamos a definição do conceito de precedência causal utilizada por Garcia [52], que é baseada na definição proposta por Lamport [77]:

Definição 2.1 Precedência Causal: Um evento e_a^α precede causalmente um evento e_b^β ($e_a^\alpha \rightarrow e_b^\beta$) se, e somente se,

- (i) $a = b$ e $\beta = \alpha + 1$; ou
- (ii) $\exists m \mid e_a^\alpha = \text{envio}(m)$ e $e_b^\beta = \text{entrega}(m)$; ou
- (iii) $\exists e_c^\gamma \mid e_a^\alpha \rightarrow e_c^\gamma \wedge e_c^\gamma \rightarrow e_b^\beta$

Por esta definição, vemos que esta relação captura eventos que possuem uma ordem definida, onde um evento e' obrigatoriamente deve ocorrer após um outro evento e . Estes eventos podem estar tanto no

mesmo processo como em processos distintos. Quando estão em processos distintos, esta dependência é criada pelo envio de mensagens entre processos.

Checkpoints globais consistentes

Checkpoints globais são considerados consistentes quando o estado formado pelos checkpoints locais de cada processo mais o canal de comunicações é consistente. Como já vimos, um fator que causa inconsistências no estado global da aplicação são as relações de precedência causais. Dizemos que um checkpoint local c de um processo p precede causalmente o checkpoint local c' de um processo p' se, e somente se, um evento e , posterior a c em p , precede um evento e' , anterior a c' em p' . Em particular, um checkpoint global consistente não pode possuir dois checkpoints locais c e c' com relação de precedência causal $c \rightarrow c'$, uma vez que c' só poderia ocorrer após a ocorrência de c e não simultaneamente.

Na verdade é possível provar uma relação muito mais forte entre checkpoints globais consistentes e relações de precedência causal, que diz que um checkpoint global é consistente se, e somente se, não existe nenhuma relação de precedência causal entre seus checkpoints constituintes. A prova pode ser encontrada em [140].

Figura 2.1: Relações de precedência entre checkpoints.

Na Figura 2.1, o checkpoint $c1$ precede causalmente $c2$, de modo que o checkpoint global $s1$ formado por $c1$ e $c2$ não é consistente. Já o checkpoint global $s2$ formado por $c3$ e $c4$ é consistente, pois não existem relações de precedência causal.

Linha de recuperação

A linha de recuperação corresponde ao checkpoint global consistente mais recente que pode ser recuperado. Este é um conceito importante, pois o objetivo da recuperação por retrocesso é reiniciar uma aplicação paralela a partir de sua linha de recuperação. Deve-se notar que esta linha de recuperação não precisa necessariamente corresponder a um estado que ocorreu durante a execução da aplicação, mas a um estado que poderia ter ocorrido. Outra utilização importante da linha de recuperação é na coleta de lixo. Uma vez determinada a linha de recuperação, pode-se eliminar todos os checkpoints que a precedem.

A dificuldade na determinação da linha de recuperação pode variar dependendo do protocolo de

checkpointing escolhido. Em alguns casos, o protocolo garante que o último *checkpoint* global é sempre consistente e, nesse caso, a determinação da linha de recuperação é trivial. Já no caso onde os processos gravam seus *checkpoints* de maneira independente, é preciso construir um grafo que captura as dependências entre estes *checkpoints* e percorre-lo até encontrar a linha de recuperação.

2.1.2 Protocolos de *checkpointing*

Protocolos de *checkpointing* de aplicações paralelas podem ser divididos em três categorias: não-coordenados, induzidos por comunicação e coordenados [46].

Protocolos não-coordenados

Protocolos não-coordenados (ou assíncronos) consistem em deixar a aplicação gerar *checkpoints* livremente. Deste modo, eles fornecem o máximo de autonomia às aplicações para que elas escolham os momentos mais convenientes para gerar checkpoints, por exemplo num ponto em que exista pouca informação que precisa ser salva.

Em compensação, os protocolos não coordenados não fornecem nenhuma garantia em relação ao avanço da linha de recuperação. No pior caso, em função das dependências causais entre os *checkpoints* salvos, pode ocorrer um efeito dominó [110], onde a aplicação é obrigada retornar ao seu estado inicial. Mas, mesmo quando isto não ocorre, ela pode ainda ser obrigada a retornar por diversos checkpoints, perdendo uma grande quantidade de computação realizada. Este é o grande problema deste tipo de protocolo e o motivo pelo qual os protocolos coordenados e induzidos por comunicação foram criados. Existem ainda outros problemas, como o fato de não haver um limite no número de *checkpoints* que precisam ser armazenados e a dificuldade na implementação de mecanismos para a determinação da linha de recuperação.

Protocolos induzidos por comunicação

Protocolos induzidos por comunicação (ou quasi-síncronos) permitem que os processos de uma aplicação paralela escolham quando gerar seus *checkpoints* locais, mas forçam estes processos a também gerar *checkpoints* extras, de modo a garantir o progresso da linha de recuperação. Os protocolos são chamados de induzidos por comunicação porque os *checkpoints* extras normalmente são gerados antes de processos enviarem ou receberem mensagens.

A idéia dos protocolos induzidos por comunicação é dar autonomia aos processos para escolher o momento de gerar seus *checkpoints* locais, de modo a realizá-los quando for mais conveniente. Este é o mesmo princípio dos protocolos não-coordenados, mas com a vantagem de que não ocorre o efeito cascata. Além disso, como não existe a necessidade de coordenação global, espera-se que estes protocolos possuam uma boa escalabilidade. Mas dependendo da quantidade e padrão de comunicação entre

os processos, o número de *checkpoints* extras pode ser bastante alto, inviabilizando sua utilização para determinados tipos de aplicações paralelas.

Existem diversos tipos de protocolos induzidos por comunicação. Estes se diferenciam em diversos pontos, como sua dificuldade de implementação, número de *checkpoints* extras gerados, quantidade de informação extra incluída nas trocas de mensagens e garantia de progressão da linha de recuperação. Uma descrição de diversos desses protocolos pode ser encontrada na tese de Garcia [52].

Protocolos coordenados

Os protocolos coordenados (ou síncronos) o extremo oposto dos protocolos não-coordenados. Nestes protocolos, é feita uma coordenação entre os processos de modo que apenas *checkpoints* globais consistentes sejam gerados. Isso é feito a partir de trocas de mensagens entre os processos, de modo que sempre que um processo gera um *checkpoint* local, todos os demais processos também geram seus *checkpoints*. Deste modo, os *checkpoints* são salvos de maneira síncrona, e por este motivo estes protocolos são também chamados de protocolos síncronos.

A grande vantagem deste tipo de protocolo é que os processos sempre são reiniciados a partir do último *checkpoint* gerado¹. Além disso, o algoritmo de coleta de lixo é trivial, uma vez que todos os *checkpoints* além do último podem ser descartados. Também não são necessários algoritmos para a determinação da linha de recuperação, uma vez que ela é composta pelos últimos *checkpoints* locais salvos.

A implementação de protocolos coordenados pode ser feita utilizando um processo coordenador que inicia o protocolo enviando mensagens a todos os processos da aplicação solicitando que estes salvem um novo *checkpoint* local. Ao receber a mensagem, os processos gravam um *checkpoint* provisório, suspendem a comunicação com outros processos e enviam uma mensagem de volta ao coordenador. Após o coordenador receber todas as respostas, este envia novas mensagens de confirmação para os processos para que estes armazenem o *checkpoint* provisório e continuem sua execução. O grande problema deste protocolo é que durante a fase de coordenação, os processos que dependem da comunicação com outros processos para continuar sua execução ficam bloqueados. Dependendo do caso, este bloqueio pode causar um atraso significativo na aplicação. Para evitar este problema, protocolos coordenados não-bloqueantes foram propostos.

Protocolos coordenados não-bloqueantes

A necessidade do bloqueio na comunicação durante a fase de coordenação em protocolos coordenados é prevenir que, durante esta fase, novas dependências entre os processos sejam criadas. Isto pode ocorrer no caso de um processo que já gravou seu *checkpoint* enviar uma mensagem a um outro processo que a recebe antes de gravar seu *checkpoint*.

¹Caso ocorra uma falha durante a fase de sincronização, pode ser necessário retornar para o penúltimo *checkpoint* salvo.

No caso de canais FIFO, pode-se utilizar o algoritmo *distributed snapshots* [29]. Neste algoritmo, um iniciador grava um *checkpoint* e envia mensagens aos demais processos requerendo que estes também gravem um novo *checkpoint*. Ao receber uma mensagem, o processo grava o *checkpoint* e repassa uma mensagem a todos os demais processos antes de enviar qualquer outra mensagem.

Podemos realizar uma modificação para que este protocolo funcione em canais que não sejam FIFO [80]. Neste caso, os processos passam a incluir, em cada mensagem enviada, o índice do último *checkpoint* salvo. Neste caso, um processo deve gerar um novo *checkpoint* sempre que receber uma mensagem com um índice maior que o do seu último *checkpoint* salvo.

Comparação entre os protocolos

Nesta seção faremos uma comparação entre os protocolos coordenados e os induzidos por comunicação. Não incluímos os protocolos não-coordenados nesta comparação porque, uma vez que estes não fornecem garantias com relação à evolução da linha de recuperação, eles são de menor interesse prático.

O principal ponto a favor dos protocolos induzidos por comunicação é que eles não necessitam de processos de sincronização globais. No caso de protocolos coordenados, a quantidade de mensagens cresce com o número de processos e, dependendo do protocolo, esse número cresce de maneira quadrática. Em função disto, em teoria, os protocolos induzidos por comunicação tem uma escalabilidade melhor com relação ao número de processos da aplicação do que os coordenados. Mas o rápido aumento de velocidade nas redes de comunicações modernas faz com que o custo extra para realizar a coordenação seja menos significativo. Além disso, nos protocolos induzidos por comunicação, são incluídos dados de controle extras nas mensagens trocadas pelos processos.

Os protocolos coordenados, por outro lado, apresentam diversas vantagens. A primeira é que a implementação do mecanismo de recuperação por retrocesso é bastante simplificada, uma vez que o protocolo sempre gera *checkpoints* globais consistentes. No caso dos protocolos induzidos por comunicação, é necessário um mecanismo para determinar a linha de recuperação baseado nas relações de precedência entre os checkpoints. Além disso, nos protocolos coordenados, o processo de eliminar *checkpoints* que não serão mais usados é bastante simples, bastando eliminar todos os checkpoints anteriores ao último *checkpoint* global.

Outro ponto favorável aos protocolos coordenados é que estes geram exatamente n *checkpoints* locais para cada *checkpoint* global gerado, onde n é o número de processos da aplicação paralela. Por outro lado, os protocolos induzidos por comunicação forçam os processos da aplicação a gerarem *checkpoints* extras, aumentando a sobrecarga do protocolo. Em alguns protocolos, o número de *checkpoints* extras gerado é ilimitado, enquanto outros garantem que serão gerados no máximo $n - 1$ *checkpoints* forçados para cada novo *checkpoint* global.

Uma possível desvantagem dos protocolos coordenados é a presença de um nó coordenador, que pode vir a ser um ponto único de falha. Mas no caso de aplicações paralelas, a falha de um nó de

processamento normalmente requer o reinício de toda a aplicação a partir do último checkpoint gerado, independentemente deste ser ou não o nó coordenador. Deste modo, na prática não existe diferença se o nó que falhou é o coordenador ou não.

Finalmente, em protocolos coordenados, pode-se ainda ter um controle sobre o período entre os *checkpoints* globais gerados. Para tal, basta ter um processo como coordenador que periodicamente inicia o protocolo de *checkpointing*. Este controle permite que se encontre um balanceamento entre o tempo de computação perdido no caso de uma falha e o custo de se executar o algoritmo de *checkpointing*. No caso de protocolos induzidos por comunicação é mais difícil obter-se este tipo de controle.

2.2 Abordagens para *checkpointing* de processos

Existem duas principais abordagens para realizar *checkpointing* de processos. A primeira, denominada *checkpointing* no nível do sistema, consiste em criar o *checkpoint* a partir de uma cópia direta do conteúdo do espaço de memória do processo. Na segunda abordagem, denominada *checkpointing* no nível da aplicação, a aplicação é responsável por fornecer os dados que serão armazenados no checkpoint.

2.2.1 *Checkpointing* no nível do sistema

Checkpointing no nível do sistema consiste em obter e armazenar o estado da aplicação diretamente a partir dos dados contidos no espaço de memória da aplicação, junto com informações de registradores e do estado do sistema operacional. Normalmente, estes dados da aplicação são salvos por um segundo processo, que inicia o processo de geração de um *checkpoint* periodicamente ou em resposta a um sinal ou mensagem externa.

Uma classe de implementações de *checkpointing* no nível de sistema requerem permissão para execução no nível do núcleo (*kernel*) do sistema operacional [95, 146]. Isto facilita amplamente o processo de *checkpointing* e a inclusão de otimizações, já que o sistema tem acesso irrestrito ao gerenciador de memória, registradores e tabelas internas ao sistema operacional. A grande desvantagem é que esta abordagem requer modificações no núcleo do sistema operacional. Além disso, o mecanismo de *checkpointing*, normalmente, precisa ser desenvolvido para uma versão específica do sistema operacional, sendo necessário que o usuário utilize apenas as versões do sistema operacional para as quais o mecanismo de *checkpointing* fornece suporte. Os *checkpoints* gerados são não-portáteis, podendo ser recuperados apenas em máquinas com a mesma arquitetura e rodando a mesma versão do sistema operacional daquela onde foram produzidos.

Existem outras implementações de *checkpointing* no nível do sistema que não necessitam de privilégios para acessar estruturas internas do núcleo do sistema operacional [82, 97]. Neste caso, modificações no núcleo não são necessárias, o que facilita bastante sua implantação. Além disso, é mais simples fazer com que o sistema funcione em diferentes versões do sistema operacional. Mas estes mecanismos

de *checkpointing* normalmente possuem limitações com relação às chamadas de sistema que o processo pode realizar. Continua também a existir uma dependência entre o mecanismo de *checkpointing* e o núcleo do sistema operacional, o que dificulta a adaptação do mecanismo para outros sistemas operacionais. Finalmente, os *checkpoints* só podem ser recuperados em máquinas com a mesma arquitetura e sistema operacional da máquina onde o *checkpoint* foi gerado, sendo que normalmente existem ainda limitações com relação às versões do sistema operacional.

Em ambos os casos, a recuperação do estado de um processo a partir de um *checkpoint* não é algo trivial. Em particular, é preciso tratar de estruturas de dados internas ao núcleo, como a lista de arquivos abertos. Se o sistema de recuperação for implementado no nível do usuário, ele não terá acesso a estas estruturas. Uma solução é interceptar as chamadas de sistema e emular estas estruturas internas do núcleo. Mas nem todas as estruturas podem ser emuladas. Outro problema é que se a falha tiver sido na máquina onde o processo executou, este deve ser reintegrado em uma outra máquina, o que também pode gerar diversas outras dificuldades. Um exemplo é quando a aplicação utilizava um arquivo local da máquina que falhou. Finalmente, é preciso recuperar os valores dos registradores, incluindo o contador de programa. A grande vantagem em se utilizar *checkpointing* no nível do sistema é que, respeitadas as restrições impostas pelo mecanismo de *checkpointing*, o processo é totalmente transparente à aplicação, podendo ser utilizado sem modificações para aplicações escritas em diferentes linguagens de programação.

2.2.2 *Checkpointing* no nível da aplicação

Em *checkpointing* no nível da aplicação, a aplicação é responsável por fornecer os dados que devem ser armazenados no *checkpoint* e por recuperar seu estado a partir dos dados presentes em um *checkpoint* [22, 66, 119]. Isto requer a instrumentação do código fonte de uma aplicação para que esta realize estas tarefas adicionais. Instrumentação de código fonte é uma técnica que permite adicionar funcionalidades a uma aplicação de modo que esta realize tarefas adicionais, como monitoramento, *logging* e persistência de dados.

Ao fornecer os dados a serem salvos, a aplicação pode prover também informações semânticas sobre estes dados. Esta informação semântica também está disponível durante o processo de recuperação, o que permite que os dados sejam salvos de modo a poderem ser recuperados por um processo executando em uma máquina de arquitetura distinta [66, 119]. Esta portabilidade dos *checkpoints* é uma importante vantagem para aplicações executando em uma grade composta por máquinas heterogêneas porque permite uma melhor utilização de recursos computacionais. Utilizando *checkpoints* não-portáteis, um processo pode reiniciar sua execução a partir de um *checkpoint* apenas se uma máquina com a mesma arquitetura e versão do sistema operacional daquela onde o *checkpoint* foi gerado estiver disponível.

A principal desvantagem de *checkpointing* no nível da aplicação é que inserir manualmente código para salvar e recuperar o estado da aplicação é um processo tedioso e sujeito a erros. Além disso, o código fonte da aplicação precisa estar disponível, o que nem sempre acontece. O primeiro problema é solucio-

nado utilizando um pré-compilador que automaticamente instrumenta o código fonte da aplicação. Mas estes pré-compiladores são específicos para uma determinada linguagem.

Outra desvantagem é que, como é a aplicação que inicia o processo de *checkpointing*, não é possível gerar *checkpoints* em pontos arbitrário da execução da aplicação. Os pontos onde os *checkpoints* podem ser gerados são definidos na instrumentação do código fonte da aplicação. Mas protocolos de *checkpointing* coordenados e induzidos por comunicação supõem que um novo *checkpoint* possa ser gerado a qualquer momento durante a execução de um processo. Deste modo, protocolos de *checkpointing* precisam ser alterados para serem utilizados com *checkpointing* no nível da aplicação. Um exemplo de adaptação de um protocolo coordenado para a utilização de *checkpointing* no nível da aplicação foi apresentado recentemente no contexto de aplicações MPI [22].

Utilizando *checkpointing* no nível do sistema evitamos estes dois problemas. *Checkpoints* podem ser gerados a qualquer momento e o processo é transparente à aplicação. Mas o fato desta abordagem gerar *checkpoints* não-portáteis, dependentes não só da arquitetura da máquina, mas também do sistema operacional, incluindo sua versão, faz com ela seja útil apenas para máquinas individuais ou aglomerados homogêneos. O aparecimento e popularização de diversos sistemas de virtualização, como o VMWare [139], Xen [11] e QEMU [12], reduzem o problema, uma vez que é possível prover um ambiente homogêneo para a execução de aplicações em diferentes máquinas. Mas nesse caso é necessário configurar o sistema de virtualização nas máquinas da grade, um processo que, utilizando os sistemas de virtualização atuais, é bastante trabalhoso e demorado.

2.2.3 Obtenção do estado de um processo no nível da aplicação

O estado de uma aplicação composta por um processo é constituído por seu espaço de memória e valores presentes em registradores do processador. Além disso, o sistema operacional mantém em seu estado interno informações relativas aquele processo, como lista de processos filhos e arquivos abertos. Para salvar o estado da aplicação, é preciso obter todos estes dados.

Em *checkpointing* no nível do sistema, estes dados normalmente são salvos por um processo ou *thread* independente. Este processo copia os dados do espaço de memória do processo para o *checkpoint*, e interage com o sistema operacional para obter os demais dados. Mas copiar os dados da memória diretamente em um arquivo de *checkpoint* é algo não portátil, dado que cada arquitetura e sistema operacional organiza a memória de modo distinto.

No caso de *checkpointing* no nível da aplicação, os dados a serem armazenados são disponibilizados pela própria aplicação. Estes dados podem ser fornecidos junto com informações semânticas, o que permite armazená-los de modo portátil. Mas parte destes dados não estão diretamente acessíveis a partir das funcionalidades fornecidas pelas linguagens de programação, de modo que é preciso obtê-los de modo indireto.

Registradores do processador

Os registradores do processador contêm o contador do programa e outros valores que são utilizados pelas instruções de processos durante sua execução. Em *checkpointing* no nível do sistema, como o processo pode ser interrompido a qualquer momento durante sua execução, é preciso salvar o valor destes registradores.

Em *checkpointing* no nível da aplicação, não é possível obter estes valores diretamente, uma vez que todo o procedimento é realizado pelo código instrumentado da própria aplicação. No caso do contador do programa, um procedimento possível é permitir que *checkpoints* sejam gerados apenas em determinados pontos da execução e armazenar em uma variável em qual ponto da execução o *checkpoint* foi salvo. Durante a reinicialização, basta continuar a execução a partir deste ponto. Já os demais registradores serão atualizados automaticamente com a execução do programa.

Pilha de execução

A pilha de execução contém dados das funções ativas durante a execução de uma aplicação. Estes dados incluem variáveis locais, parâmetros de funções, endereços de retorno e outras informações de controle. A estrutura desta pilha e as informações de controle são específicas para cada arquitetura e sistema operacional. Conseqüentemente, se as informações contidas na pilha forem copiadas diretamente em um checkpoint, este só poderia ser recuperado numa máquina de mesma arquitetura e sistema operacional do original.

O código da aplicação não tem acesso direto ao conteúdo da pilha de execução, que é criada dinamicamente durante a execução de um processo, à medida que métodos são chamados e variáveis locais são criadas. Deste modo, só é possível salvar em um *checkpoint* parte do conteúdo presente na pilha de execução. Surpreendentemente, é possível recriar esta pilha apenas com a lista ordenada de funções empilhadas no momento da criação do *checkpoint* e os valores de parâmetros e variáveis locais. Como detalhamos na Seção 3.4.2, podemos reconstruir a pilha simplesmente chamando as funções presentes nesta lista e atribuindo os valores corretos a parâmetros e variáveis locais. As demais informações na pilha são automaticamente inseridas pelo ambiente de execução assim que as funções são chamadas.

Ponteiros e memória dinâmica

É necessário salvar também o conteúdo da memória alocada dinamicamente. Neste caso também não é possível acessar as estruturas de alocação de memória diretamente do código da aplicação, de modo que é necessário armazenar separadamente uma tabela com os blocos de memória alocados pela aplicação. Esta tabela mantém um mapeamento entre ponteiros presentes na aplicação e as faixas de memória alocadas dinamicamente.

No momento de salvar ponteiros, é necessário um cuidado extra, uma vez que estes apontam para

endereços específicos a uma execução particular. Além disso, se quisermos que o *checkpoint* gerado seja portátil, é necessário que estes endereços sejam convertidos para um formato independente da arquitetura.

Outra dificuldade é a existência de ponteiros com múltiplos níveis de indireção. Neste caso, é necessário percorrer todos os níveis de indireção, convertendo os endereços referenciados em cada nível. É preciso cuidado para não salvar um mesmo endereço de memória mais de uma vez, evitando gerar *checkpoints* de tamanho maior que o necessário.

Estado interno do sistema operacional

O sistema operacional armazena informações sobre todos os processos executando em uma máquina. Estas informações incluem a lista de arquivos abertos, de processos filhos e *buffers* de comunicação. O mecanismo de *checkpointing* utilizado deve salvar este estado interno do sistema operacional ou proibir determinadas chamadas de sistema.

Quando usamos *checkpointing* no nível do sistema com acesso ao núcleo do sistema operacional, é simples obter estas informações. Nas demais abordagens de *checkpointing*, é necessário obter este estado indiretamente. Uma solução para este problema é criar funções do tipo “wrapper” para algumas chamadas de sistema que modificam o estado do sistema operacional. Estas funções armazenam dados extras, de modo que a parte do estado do sistema operacional que interessa à aplicação possa ser armazenada e posteriormente recuperada.

2.3 Execução de aplicações paralelas em arquiteturas heterogêneas

Para realizar a execução de aplicações paralelas em máquinas de arquitetura heterogênea, precisamos considerar a diferença na representação dos diferentes tipos de dados nestas arquiteturas. Um exemplo típico é a ordenação de bytes em tipos primitivos, denominada *endianness*. É necessário realizar a conversão na representação dos dados sempre que processos executando em arquiteturas se comunicam.

2.3.1 O modelo de programação BSP

O modelo BSP foi introduzido por Leslie Valiant como um modelo que integra arquitetura e software [136]. Um computador abstrato BSP é composto por uma coleção de processadores virtuais com memória local, interligados por um rede cujas únicas propriedades relevantes são o tempo para realizar uma barreira de sincronização e a velocidade com a qual dados acessados aleatoriamente são transmitidos. Uma computação BSP é composta por uma sequência de super-passos. Cada super-passo é composto por uma fase de computação e uma de comunicação, que termina com uma barreira de sincronização.

Uma vantagem do modelo BSP sobre outras abordagens para programação paralela, como PVM e MPI, é sua interface simplificada, o que facilita a implementação de bibliotecas BSP e de aplicações paralelas baseadas no modelo BSP. Outra vantagem é a previsibilidade do desempenho, que é analisado supondo-se que em uma unidade de tempo um processador pode realizar uma operação em um dado disponível em sua memória local. Esta análise é baseada em 3 parâmetros: o número de processadores virtuais (P), a taxa entre a velocidade da rede e do processador (G) e o tempo necessário para realizar a barreira de sincronização entre todos os processos (L). Finalmente, como o modelo BSP requer sincronizações periódicas entre os processos, protocolos coordenados de checkpointing podem ser facilmente implementados.

Mas este modelo apresenta também desvantagens, como o fato de necessitar de sincronizações periódicas entre seus processos. Dependendo de quão frequentes forem estas sincronizações, o desempenho da aplicação pode ser comprometido. Além disto, nem todas as aplicações paralelas podem ser implementadas eficientemente utilizando o modelo BSP. Finalmente, os parâmetros G e L citados anteriormente não podem ser facilmente calculados no ambiente dinâmico de uma grade oportunista.

Diversas implementações do modelo BSP foram desenvolvidas, incluindo a BSPlib de Oxford [62], a PUB [21], a BSPlib do InteGrade [57] e a BSP-G do Globus [132]. Apesar de todas seguirem o mesmo modelo de programação, existem pequenas diferenças nas APIs fornecidas por diferentes implementações.

Utilizando a API da BSPlib do InteGrade, o trecho BSP da execução de uma aplicação se inicia após todos os processos da aplicação chamarem o método `bsp_begin`, que realiza a sincronização dos processos e atribui a cada processo um identificador único *pid*. Um dos processos, que recebe *pid* = 0 e que denominamos *processo zero*, fica responsável pela coordenação das etapas de sincronização no início da execução e no final de cada superpasso. O término de superpasso ocorre quando todos os processos chamam o método `bsp_sync`, que implementa a barreira de sincronização e garante que toda a comunicação entre os processos realizada durante o superpasso seja efetivada. O trecho BSP da execução termina quando todos os processos chamam `bsp_end`, que realiza a sincronização da execução dos processos.

O modelo BSP possui dois modos para comunicação entre processos, denominados *Direct Remote Memory Access* (DRMA), que implementa a funcionalidade de uma memória compartilhada distribuída, e *Bulk Synchronous Message Passing* (BSMP), que permite a comunicação por troca de mensagens. Quando utilizamos DRMA, os processos da aplicação registram endereços de memória locais como endereços virtuais compartilhados, através da função `bsp_push_reg`. Uma vez registrados, outros processos podem ler e escrever nestes endereços através dos comandos `bsp_get` e `bsp_put`. No caso de comunicações baseadas em BSMP, mensagens são enviadas por um processo a outro por meio da função `bsp_send`. Estas mensagens são armazenadas em um *buffer* pelo processo receptor e podem ser acessadas utilizando o comando `bsp_move`.

No exemplo abaixo, ilustramos uma aplicação que utiliza a BSPlib do InteGrade. Esta aplicação,

composta por 4 processos, lê uma sequência de arquivos e, para cada arquivo, calcula a média dos valores nele contido. Cada processo é responsável por ler 1/4 de cada arquivo, calcular a média dos valores contidos em sua parte e enviar ao processo zero o valor desta média. Seleccionamos uma aplicação bastante simples para facilitar o entendimento do modelo BSP. Outros exemplos de aplicações BSP, utilizadas em ambientes acadêmicos, podem ser encontrados em [135].

```
1 int main(int argc, char **argv) {
2
3     int n_processos = 4;
4     int n_arquivos = 100;
5     int pid;
6     double media;
7     double *lista_de_medias;
8
9     lista_de_medias = (double *)malloc(n_processos*sizeof(double));
10
11     bsp_begin( n_processos );
12     bsp_push_reg( &lista_de_medias, n_processos*sizeof(double) );
13     pid = bsp_pid();
14
15     for (int i=0; i<n_arquivos; i++) {
16
17         media = calculaMedia(pid, n_processos);
18
19         if (pid != 0)
20             bsp_put (0, &media, &lista_de_medias, pid, sizeof(double));
21
22         bsp_sync();
23
24         if (pid == 0)
25             for (int p=1; p<n_processos; p++)
26                 media += lista_de_medias[p];
27         media /= n_processos;
28
29         printf("A media dos valores contidos no arquivo %d e %f", i, media);
30     }
31
32     bsp_end();
33 }
```

Como podemos ver, o processo inicializa a BSPlib chamando o método `bsp_begin`, na linha 11. Em seguida, o processo registra o endereço do vetor *lista_de_medias*, de modo a torná-lo visível aos demais processos da aplicação. Na linha seguinte, cada processo obtém o seu identificador único *pid*, chamando o método `bsp_pid`.

A parte principal da aplicação está contida no laço `for`, na linha 15, onde cada processo chama o método `calculaMedia`, que recebe como entrada o `pid` do processo e `n_processos`, lê a parte do arquivo atribuída àquele processo e devolve a média dos valores lidos. Após calcular a média, cada processo, exceto o 0, escreve o valor da média de sua parte do arquivo no vetor `lista_de_medias`, localizado na memória local do processo 0. Em seguida, na linha 22, os processos chamam `bsp_sync`, de modo a finalizar o superpasso e permitir que a BSPlib realize a escrita remota dos dados. Após o término da sincronização, o processo 0 lê os valores escritos pelos demais processos e realiza o cálculo da média dos valores contidos no arquivo.

Após o término do cálculo da média de todos os arquivos, os processos chamam `bsp_end` de modo a finalizar a execução da aplicação BSP.

2.3.2 Comunicação portátil entre processos

Apesar de existirem diversas implementações do modelo BSP para diferentes arquiteturas, todas requerem que os processos pertencentes a uma dada aplicação BSP devem executar em nós de mesma arquitetura. O problema para a comunicação em arquiteturas heterogêneas é que suas APIs não proporcionam uma maneira de especificar o tipo de dado sendo transmitido. Os dados transmitidos são tratados como seqüências de bytes.

Para permitir a comunicação entre processos executando em máquinas de arquiteturas distintas, estendemos a API da BSPlib adicionando, a alguns métodos, um parâmetro extra que descreve o tipo de dado que está sendo passado a esta função. Mas para utilizar esta API estendida, o programador teria que modificar suas aplicações para que fizessem uso dessas novas funções. Para evitar este problema, utilizamos um pré-compilador que automaticamente instrumenta o código fonte da aplicação. Informações sobre o tipo de dado passado para as funções são obtidas pelo pré-compilador, que as utiliza para inserir o novo parâmetro nos métodos modificados. Deste modo, uma aplicação escrita para a BSPlib original é automaticamente transformada para utilizar a versão estendida da BSPlib.

Uma limitação imposta por esta abordagem é que o programador deve evitar realizar conversões (*casts*) arbitrárias em dados que serão compartilhados com outros processos da aplicação. Isto porque a versão estendida da biblioteca BSPlib utiliza informações dos tipos de dados compartilhados no momento de realizar a conversão entre as representações destes dados nas diferentes arquiteturas.

2.4 Armazenamento de checkpoints

Grades oportunistas são tipicamente compostas por um conjunto de aglomerados de máquinas, correspondentes à organização física destas máquinas. Aplicações normalmente são executadas em um único aglomerado, mesmo no caso de aplicações paralelas. O mecanismo de *checkpointing* gera *checkpoints* periódicos que precisam ser armazenados. Estes *checkpoints* podem ter tamanhos da ordem de

gigabytes no caso de aplicações paralelas.

Poderíamos armazenar os *checkpoints* em um repositório dedicado daquele aglomerado, mas isto gera a necessidade de uma máquina reservada para esta tarefa. Um servidor central poderia ser um gargalo no momento em que os *checkpoints* locais de cada processo de uma aplicação paralela fossem transferidos simultaneamente, além de ser um ponto único de falha no sistema. O uso de um aglomerado já conectado a um sistema de arquivos de rede, como o NFS, dispensa um servidor dedicado para checkpoints, mas sofre dos mesmos problemas de desempenho e tolerância a falhas.

Uma solução seria armazenar os *checkpoints* no próprios nós de processamento da grade. Mas a máquina onde um processo é executado não pode ser utilizada para armazenar um *checkpoint* daquele processo de modo confiável, pois normalmente as falhas na execução do processo são causadas justamente porque a máquina onde ele executava ficou indisponível. Neste caso, o *checkpoint* ficaria inacessível para reiniciar a aplicação em outra máquina. Os *checkpoints* poderiam ser então armazenados em outros nós daquele aglomerado. Mas estes nós também não são dedicados e podem ficar indisponíveis a qualquer momento.

Deste modo, ao utilizarmos os nós de uma grade oportunista para o armazenamento de checkpoints, devemos utilizar alguma forma de redundância para garantir que os *checkpoints* possam ser recuperados. No caso de aplicações sequenciais, uma estratégia simples é salvar uma única cópia do checkpoint, com cada *checkpoint* sendo armazenado num nó diferente. Assim, mesmo que o último *checkpoint* não esteja disponível, *checkpoints* anteriores podem ser utilizados.

A seguir, comparamos diversas estratégias para o armazenamento remoto de *checkpoints* em repositórios não-dedicados. Estas estratégias são direcionadas a aplicações paralelas, de modo que cada *checkpoint* global gerado é composto por diversos *checkpoints* locais. Para que um *checkpoint* global possa ser utilizado, é necessário que todos os *checkpoints* locais estejam disponíveis.

2.4.1 Estratégias de armazenamento

Ao avaliar estratégias de armazenamento para *checkpoints* de aplicações paralelas, precisamos considerar diversas propriedades, como escalabilidade, custo computacional e tolerância a falhas. Outro ponto importante a ser considerado é decidir em quais repositórios os checkpoints devem ser armazenados. É possível distribuir os dados entre os nós executando a aplicação e em outros nós do aglomerado. Analisamos diversas estratégias de armazenamento, como replicação e codificação de dados, com relação aos critérios acima.

Paridade de dados

Esta estratégia consiste em armazenar apenas uma cópia do *checkpoint* com a adição de informação de paridade [70]. Para calcular a paridade de um checkpoint, é preciso primeiro dividi-lo em m fragmentos e então calcular a paridade sobre estes fragmentos. Um *checkpoint* C de tamanho n é dividido em m

fragmentos U_k de tamanho n/m :

$$C = (U_0, U_1, U_2, \dots, U_m)$$

$$U_k = (u_0^k, u_1^k, \dots, u_{n/m}^k), 0 \leq k < m$$

Os elementos p_i , $0 \leq i < n/m$ do fragmento P , contendo a paridade dos demais fragmentos, são dados por:

$$p_i = (u_i^0 \oplus u_i^1 \oplus \dots \oplus u_i^m), 0 \leq i < n/m,$$

onde $\oplus$ representa a operação de ou-exclusivo. Os fragmentos U_i e P de cada processo são então distribuídos para o armazenamento em outros nós.

Utilizando a estratégia de paridade de dados, é possível reconstruir um fragmento arbitrário U_k combinando o conteúdo dos demais fragmentos com a informação de paridade contida no fragmento P , através das equações:

$$U_k = (u_0^k, u_1^k, \dots, u_{n/m}^k), 0 \leq k < m$$

$$u_i^k = (u_i^0 \oplus u_i^1 \oplus \dots \oplus u_i^{k-1} \oplus p_i \oplus u_i^k \oplus \dots \oplus u_i^m), 0 \leq i < n/m$$

Se supormos que cada fragmento será armazenado em um nó diferente, vemos que esta estratégia provê tolerância à falha simultânea de um único nó. O tamanho do vetor de paridade P é inversamente proporcional ao número de fragmentos, de modo que a utilização de mais fragmentos requer menos espaço de armazenamento para os checkpoints. Por outro lado, por utilizar mais nós de armazenamento, é maior a probabilidade de ocorrer falhas simultâneas em dois ou mais nós. Além disso, é preciso criar mais conexões para realizar a transferência de arquivos.

Uma maneira bastante eficiente de implantar esta estratégia de armazenamento é utilizando os nós onde a aplicação paralela está sendo executada para armazenar os fragmentos dos *checkpoints* locais. Caso ocorra uma falha em um dos nós da aplicação, é possível reiniciá-la a partir da informação contida nos demais nós. Por outro lado, se uma segunda falha ocorrer, não será mais possível recuperar o estado da aplicação até que pelo menos uma das duas máquinas que falharam fique disponível novamente.

A vantagem de utilizar paridade, quando comparado a outras estratégias de armazenamento, é que seu cálculo é bastante eficiente, necessitando de apenas $O(n)$ operações ou-exclusivo, onde n é o tamanho do vetor a ser codificado. Sua desvantagem é que a falha de dois ou mais nós contendo fragmentos do *checkpoint* é o suficiente para tornar o *checkpoint* irrecuperável.

Algoritmo de dispersão de informação (IDA)

O algoritmo de dispersão de informação (IDA) [101] clássico, desenvolvido por Rabin, permite gerar uma codificação ótima dos dados com relação ao uso de memória. Usando IDA, é possível codificar um vetor U de tamanho n em $m + k$ vetores codificados de tamanho n/m , com a propriedade de que o vetor

original U pode ser reconstruído utilizando apenas m vetores codificados. Utilizando esta codificação, é possível obter diferentes níveis de tolerância a falhas ajustando os valores de m e k . Na prática, é possível tolerar k falhas com uma sobrecarga de apenas $k * n/m$ elementos. Denominamos esta classe de codificação em fragmentos redundantes como códigos de “rasura” (*erasure codes*).

Este algoritmo requer o uso de operações matemáticas sobre um campo de Galois $GF(q)$, um campo finito de q elementos, onde q é um número primo ou uma potência p^x de um número primo p . Quando utilizamos $q = p^x$, operações aritméticas sobre o campo são realizadas representando os números como polinômios de grau $x - 1$ e coeficientes em $[0, p - 1]$. Somas são calculadas utilizando operações de ou-exclusivo, enquanto multiplicações são calculadas multiplicando os polinômios módulo um polinômio irredutível de grau x .

O algoritmo requer a geração de $m + k$ vetores linearmente independentes α_i de tamanho m . Estes vetores podem ser facilmente obtidos escolhendo n valores distintos a_i , $0 \leq i < n$ e definindo $\alpha_i = (i, a_i, \dots, a_i^{n-1})$, $0 \leq i < n$. Estes vetores são então organizados como uma matriz G definida como:

$$G = [\alpha_0^T, \alpha_1^T, \dots, \alpha_{m+k}^T]$$

Dividimos então o vetor U em n/m palavras de informação U_i de tamanho m e geramos n/m palavras codificadas V_i de tamanho $m + k$, onde:

$$V_i = U_i \times G$$

Finalmente, os $m + k$ vetores codificados E_i , $0 \leq i < m + k$ são definidos por:

$$E_i = (V_0[i], V_1[i], \dots, V_{n/m}[i])$$

Para recuperar as palavras de informação originais U_i , temos que recuperar m dos $m + k$ fragmentos codificados. Em seguida, construímos as palavras V'_j , que são equivalentes às palavras codificadas V_i , mas contêm apenas os componentes dos m fragmentos recuperados. De modo similar, construímos uma matriz G' , contendo apenas elementos relativos aos fragmentos recuperados. Podemos então recuperar as palavras de informação originais U_i multiplicando as palavras codificadas V'_j com a matriz inversa de G' :

$$U_i = V'_i \times G'^{-1},$$

O maior problema desta abordagem é que o processo de codificação requer $O((m + k) * n * m)$ passos e o de decodificação $O(n * m * m)$ passos. O processo inclui ainda a inversão de uma matriz $m \times m$, mas m normalmente é um número pequeno.

Malluhi e Johnston [87] propuseram um algoritmo que melhora a complexidade de codificação para $O(n * m * k)$ e melhora também a de decodificação. Eles mostraram que podemos diagonalizar as primeiras m colunas de G e ainda obter um algoritmo válido. Conseqüentemente, os primeiros m campos

das palavras codificadas V_i envolvem uma simples cópia de dados. O processo de multiplicação de matriz só é necessário para os últimos k campos.

As principais vantagens de utilizar IDA são que este algoritmo permite definir o nível de tolerância a falhas desejado e que a sobrecarga gerada sobre o tamanho do arquivo é pequena. Para uma aplicação composta por 10 nós, se configurarmos m e k como 4, é possível tolerar falhas simultâneas em quaisquer 4 de 8 máquinas com apenas o dobro de dados do vetor inicial. A desvantagem do uso desta abordagem é sua alta complexidade computacional.

Replicação de dados

A replicação total de dados consiste em armazenar réplicas completas dos *checkpoints* gerados. Se uma das réplicas se torna inacessível, podemos utilizar outra que esteja disponível. A vantagem desta abordagem é que ela é de simples implementação e não possui o custo computacional requerido para calcular paridades ou codificar fragmentos.

Em compensação, é necessário transferir e armazenar grandes quantidades de dados. Para garantir tolerância contra a falha de um único nó de armazenamento, é necessário salvar duas cópias completas do checkpoint. No caso de um arquivo de checkpoint de 100MB, é necessário transferir e armazenar 200MB de dados.

Códigos de correção de erros

Códigos de correção de erros permitem a localização e correção de erros em uma sequência de dados. Um dos códigos mais comuns, Reed-Solomon [96], permite a correção de $k/2$ erros em bits, onde k é o número de bits extras. O algoritmo de dispersão de informação pode corrigir k bits de erros utilizando os mesmos k bits. A diferença se deve ao fato que, ao utilizar o algoritmo de dispersão de informação, temos de saber antecipadamente sobre a presença de erros e sua localização. A codificação de Reed-Solomon pode detectar e corrigir erros em locais arbitrários.

2.5 Resumo

Neste capítulo introduzimos o conceito de *checkpointing*, os protocolos de *checkpointing* para aplicações paralelas e as abordagens para obter o estado de um processo. Apresentamos também o modelo de programação BSP e diversas estratégias para o armazenamento de *checkpoints* em máquinas não-dedicadas.

Como veremos no próximo capítulo, onde abordaremos a implementação do sistema de tolerância a falhas baseado em *checkpointing* do InteGrade, decidimos utilizar *checkpointing* no nível da aplicação e um protocolo de *checkpointing* coordenado. A escolha do *checkpointing* no nível da aplicação foi feita com base na heterogeneidade dos computadores disponíveis numa grade oportunista. Já a escolha do

protocolo coordenado foi feita porque, além das vantagens apresentadas, o middleware de grade utilizado neste trabalho dá suporte ao modelo de programação BSP. Este modelo requer a sincronização periódica entre seus processos, de modo que o protocolo de *checkpointing* pode se utilizar desta sincronização para gerar *checkpoints* globais consistentes.

Capítulo 3

Tolerância a Falhas de Aplicações no Sistema InteGrade

Como vimos no Capítulo 2, um mecanismo de tolerância a falhas é essencial para permitir a execução de aplicações paralelas de longa duração numa grade oportunista. Neste capítulo, descrevemos nossa implementação do mecanismo de recuperação por retrocesso baseada em *checkpointing* do InteGrade. Este mecanismo inclui um pré-compilador e uma biblioteca de *checkpointing* para salvar estados portáteis de aplicações sequenciais e de aplicações paralelas do tipo BSP. Além disso, o mecanismo possui diversos módulos que realizam o gerenciamento da execução de aplicações, incluindo a detecção de falhas na execução e a reinicialização destas aplicações a partir de um *checkpoint*.

Iniciamos o capítulo apresentando trabalhos relacionados da área de *checkpointing* e tolerância a falhas. Em seguida, na Seção 3.2, discutimos a infra-estrutura para recuperação por retrocesso no InteGrade. Na Seção 3.3, descrevemos a implementação dos principais módulos associados ao processo de recuperação por retrocesso. Na Seção 3.4, descrevemos com detalhes os processos de obtenção, armazenamento e recuperação do estado de uma aplicação, realizado pelo pré-compilador e biblioteca de *checkpointing*. Finalmente, na Seção 3.5, descrevemos as alterações que realizamos na biblioteca BSP para permitir salvar o estado de aplicações BSP e sua execução em nós de arquitetura heterogênea.

3.1 Trabalhos relacionados

Existe uma abrangente literatura nas áreas de *checkpointing* e armazenamento distribuído de dados em aglomerados. Nesta seção descrevemos alguns trabalhos relacionados.

3.1.1 Checkpointing no nível da aplicação

Porch [119] é um pré-compilador que instrumenta o código fonte de uma aplicação em C de modo que esta passe a gerar *checkpoints* portáteis. No entanto, ele não fornece suporte a aplicações paralelas. Além disso, Porch requer conhecimento detalhado sobre a representação de dados e seu alinhamento em estruturas para cada compilador das arquiteturas a que fornece suporte. Deste modo, implementar suporte a uma quantidade razoável de sistemas operacionais, arquiteturas e compiladores torna-se uma tarefa bastante árdua. Na nossa implementação, necessitamos apenas da representação de tipos primitivos de dados. Finalmente, no momento de gerar um *checkpoint*, Porch desconstrói e reconstrói toda a pilha de execução do processo, de modo a obter os endereços de variáveis locais e lista de funções na pilha de execução, enquanto na nossa abordagem estes dados são obtidos dinamicamente durante a execução da aplicação.

Karablieh *et al* [66] propõem a utilização de funções de entrada e saída da linguagem de programação para salvar e recuperar dados de maneira portátil. A vantagem desta abordagem é que não é necessário escrever rotinas para a conversão de dados entre diferentes arquiteturas. A desvantagem é que este processo é muito mais lento e gera *checkpoints* maiores, uma vez que dados binários são convertidos em texto. Além disso, a abordagem utilizada requer um nível extra de indireção para todos os acessos a ponteiros *durante* a execução da aplicação, gerando uma grande sobrecarga no caso de aplicações que utilizam muitos ponteiros.

Chanchio e Sun [28] desenvolveram uma metodologia para salvar o estado de uma aplicação numa representação independente de arquitetura de modo a permitir sua recuperação numa máquina de arquitetura distinta. Para tal, são utilizadas diversas estruturas que armazenam informações necessárias para salvar os dados em um checkpoint. Durante a criação do checkpoint, os dados primitivos são convertidos para uma representação intermediária. A abordagem para salvar os dados de modo portátil é similar à nossa, mas difere no tratamento e representação independente de arquitetura dos blocos de memória alocados dinamicamente pela aplicação.

O ambiente de migração de objetos distribuídos (DOME) [14] fornece um ambiente que permite que aplicações C++ tenham o seu estado salvo de maneira portátil. É disponibilizado um pré-compilador que instrumenta o código fonte da aplicação para salvar seu estado periodicamente. Dados primitivos são convertidos para a representação intermediária XDR [116]. Mas neste trabalho, é necessário que as aplicações sejam escritas utilizando o modelo de programação do DOME, o que limita bastante sua aplicabilidade. Em particular, o programador precisa utilizar estruturas de dados específicas do DOME.

Kasbekar *et al* [69] propõem o uso de reflexão em tempo de compilação e de execução para realizar o retrocesso seletivo de programas concorrentes em C++. O protocolo de meta-objetos do OpenC++ [32] é utilizado para gerar métodos para salvar o estado de classes e depois recuperá-los. Durante a execução, as comunicações entre objetos são monitoradas de modo que durante o retrocesso apenas as *threads* com dependências entre si precisam ser retrocedidas. Mas não são fornecidos detalhes de como a recuperação

do estado da aplicação é realizada e como problemas como polimorfismo são tratados. Além disso, foi realizado um estudo experimental bastante simplista.

3.1.2 *Checkpointing* de aplicações paralelas

Hill *et al* [61] propuseram um mecanismo de *checkpointing* transparente para aplicações BSP que utilizem a BSPLib de Oxford. A idéia é utilizar *checkpointing* para prover tolerância a falhas e balanceamento de carga para aplicações paralelas sendo executadas em um aglomerado. Esta biblioteca utiliza *checkpointing* no nível do sistema, já que o foco do sistema é em aglomerados homogêneos.

Bronevetsky *et al* [22] propuseram a utilização de *checkpointing* no nível da aplicação para aplicações MPI. Para tal, é desenvolvido um protocolo de *checkpointing* coordenado que funciona com *checkpointing* no nível da aplicação. Um pré-compilador que instrumenta o código fonte de aplicações C também é fornecido. A biblioteca de *checkpointing* implementada por Bronevetsky *et al* utiliza um gerenciador de memória próprio, de modo que endereços de memória são salvos diretamente no checkpoint, sem a necessidade de conversão para *offsets*. Por outro lado, os *checkpoints* gerados não são portáteis. Como o foco do trabalho são aplicações executando em aglomerados homogêneos, a portabilidade não foi considerada como um requisito.

Diversas propostas para o uso de *checkpointing* para migração de aplicações PVM e MPI foram publicadas [1, 25, 117]. Em comum, estes trabalhos utilizam *checkpointing* no nível do sistema e a ênfase é na migração de processos de aplicações executando em aglomerados homogêneos. Em nosso trabalho, o foco é na utilização de máquinas heterogêneas. Para tal, fazemos o uso de *checkpointing* no nível da aplicação. Além disso, nosso mecanismo fornece suporte a aplicações BSP, enquanto os demais dão suporte aos modelos PVM e MPI.

3.1.3 *Checkpointing* em grades computacionais

Recentemente, pesquisas na área de tolerância a falhas para aplicações paralelas em Grades foram publicadas. O MPICH-GF [143] provê *checkpointing* transparente ao usuário para aplicações MPI executando sobre o middleware Globus [48], utilizando *checkpointing* no nível do sistema e um protocolo coordenado para a sincronização dos processos.

Condor [81] possui um mecanismo de *checkpointing* no nível do sistema para aplicações que possuem apenas um processo [82]. Este sistema foi posteriormente estendido para salvar o estado de aplicações paralelas PVM [100]. Um protocolo de *checkpointing* coordenado é utilizado para gerar *checkpoints* globais consistentes. Estes checkpoints são salvos em servidores de *checkpoints*. Outro mecanismo de *checkpointing* para aplicações PVM executando sobre o Condor é apresentado em [74]. Este mecanismo também utiliza *checkpointing* no nível do sistema, um protocolo de *checkpointing* coordenado e armazena os *checkpoints* gerados em um servidor de checkpoints.

O ponto comum destes sistemas é a utilização de *checkpointing* no nível do sistema, direcionado a aglomerados homogêneos. Nosso sistema utiliza *checkpointing* no nível da aplicação, o que nos permite trabalhar com máquinas heterogêneas de uma grade oportunista. Além disso, os sistemas acima supõem a existência de servidores para o armazenamento de checkpoints. Nosso sistema coloca ênfase na utilização dos nós compartilhados da grade para armazenar os *checkpoints* da aplicação. Finalmente, nenhum dos sistemas acima fornece suporte a aplicações BSP.

3.1.4 Estratégias de armazenamento

Existem diversos trabalhos na área de armazenamento distribuído de dados que utilizam codificação de dados para melhorar a tolerância a falhas e segurança. Diferentemente de nosso trabalho, estes tratam de repositórios de dados dedicados.

Malluhi e Johnston [87] utilizam uma versão otimizada do algoritmo de dispersão de informação (IDA) [101]. Eles comparam analiticamente este algoritmo com um processo de codificação utilizando paridade em duas dimensões. Em nosso trabalho, realizamos uma avaliação experimental dos diferentes tipos de codificação para realizar o armazenamento de *checkpoints* de aplicações paralelas, incluindo sua influência sobre o tempo de execução destas aplicações, além de focarmos em repositórios não dedicados.

Alon *et al* [3] desenvolveram estratégias para permitir o armazenamento e recuperação de dados mesmo quando ocorrem falhas em metade do sistema. Eles utilizam IDA para melhorar a recuperação dos dados e adicionam informações de espalhamento (*hash*) para garantir sua integridade. De modo similar, Garay *et al* [51] também tratam do problema de armazenamento e recuperação de informação utilizando IDA e técnicas criptográficas. Ellard *et al* [45] avaliam o armazenamento distribuído de dados de forma segura, mas trabalham com objetos de dados imutáveis. Diferentemente de nosso trabalho, os três sistemas citados possuem um escopo muito maior, fornecendo um suporte mais abrangente para o armazenamento de dados em geral. Por outro lado, implementamos suporte a diversas estratégias de armazenamento, focadas no armazenamento de *checkpoints* de aplicações paralelas em repositórios de dados não-dedicados.

Existem ainda trabalhos específicos para a área de armazenamento distribuído de *checkpoints* de aplicações paralelas. Sobe [114] analisa a utilização de duas técnicas diferentes de paridade para armazenar *checkpoints* em sistemas distribuídos, apresentando um estudo analítico para compará-las. Em nosso trabalho, o estudo realizado é experimental. Além disso, analisamos outras estratégias de armazenamento, como replicação e codificação utilizando IDA.

Plank *et al* [98] propõem o uso de *checkpointing* sem disco (*diskless checkpointing*), onde *checkpoints* são armazenados na memória volátil do sistema, eliminando assim a sobrecarga do uso de um meio de armazenamento estável. De modo similar a nosso trabalho, os autores analisam um cenário onde os dados são armazenados nos nós de processamento e em um ou mais nós de *backup*. Mas o foco do trabalho é na comparação de *checkpointing* com e sem disco e a comparação se restringe ao uso de

informações de paridade.

Finalmente, Pruyne e Livny [100] realizaram estudos sobre o uso de múltiplos servidores de *checkpoints* para armazenar *checkpoints* de aplicações paralelas executando sobre o Condor. Mas os experimentos realizados ficaram restritos à comparação do uso de 1 ou 2 servidores de checkpoints dedicados.

3.2 Visão geral de recuperação por retrocesso no InteGrade

O InteGrade [55] é um projeto que envolve diversas universidades e visa o desenvolvimento de um middleware inovador para grades computacionais. O objetivo do projeto é permitir a utilização do poder computacional ocioso de computadores pessoais para a execução de aplicações de longa duração. O InteGrade possui uma arquitetura orientada a objetos, sendo construído sobre o padrão CORBA [94] de objetos distribuídos. Um dos principais requisitos do InteGrade é o de garantir a manutenção da qualidade de serviço dos usuários que compartilham seus recursos com o sistema de Grade. O InteGrade é disponibilizado como software livre e pode ser obtido a partir da página do InteGrade na Internet (<http://www.integrate.org.br>).

Apresentamos a seguir a arquitetura do InteGrade. Em seguida, descrevemos os módulos e bibliotecas adicionados à arquitetura do InteGrade para permitir o suporte a tolerância a falhas utilizando a recuperação por retrocesso.

3.2.1 Arquitetura do InteGrade

Uma grade InteGrade é composta por uma coleção de aglomerados, organizados em uma rede hierárquica, onde cada aglomerado é tipicamente composto por algumas dezenas de máquinas localizadas numa rede local. O objetivo desta divisão em dois níveis é favorecer a escalabilidade do sistema. Nesta seção apresentamos a organização de um aglomerado InteGrade.

Cada aglomerado do InteGrade possui um nó responsável por gerenciar o aglomerado, chamado *nó de gerenciamento*, e diversos nós que compartilham recursos com a grade, que denominamos *nós provedores de recursos*. O nó de gerenciamento deve estar sempre disponível e normalmente corresponde ao servidor de uma rede local. Na Figura 3.1, apresentamos os principais módulos presentes em cada tipo de nó. Temos ainda o nó de usuário, a partir do qual um usuário pode submeter aplicações para execução na Grade. Estas categorias para os nós não são exclusivas, por exemplo podemos ter uma máquina que é nó provedor de recursos e nó de usuário simultaneamente.

O **ASCT** (*Application Submission and Control Tool*) é a ferramenta que permite que usuários da grade realizem requisições de execução de aplicações, controlem a execução destas aplicações e visualizem seus resultados. Esta ferramenta pode ser executada em qualquer máquina, independente desta compartilhar recursos com a Grade ou não. O **Portal** permite que usuários da grade realizem, utilizando um portal na Internet, as mesmas tarefas que podem ser realizadas com o ASCT. A vantagem do portal

Figura 3.1: Arquitetura de um aglomerado do InteGrade

é que usuários podem acessar remotamente o InteGrade a partir de qualquer máquina conectada à Internet utilizando um navegador, sem a necessidade de instalação de um software cliente específico do InteGrade.

O **GRM** (*Global Resource Manager*) é o módulo responsável pelo gerenciamento dos recursos de um aglomerado e pelo escalonamento da execução de aplicações, enquanto o **LRM** (*Local Resource Manager*) gerencia os recursos de uma única máquina.

O GRM mantém uma lista dos LRMs ativos e, ao receber requisições para execução de aplicações, escolhe um LRM que atenda às necessidades da aplicação. Para tal, assim que um LRM é inicializado, este localiza o GRM de seu aglomerado e se registra, enviando informações sobre características e recursos oferecidos na máquina em que reside, como velocidade do processador, quantidade de memória RAM e sistema operacional. Além disso, o LRM envia periodicamente ao GRM informações contendo a quantidade de recursos disponíveis para utilização na máquina.

O LRM recebe requisições de execução do GRM e determina se estas requisições podem ser aceitas. Em caso afirmativo, um novo processo é criado para realizar a execução. Caso contrário, o LRM notifica o GRM que a execução não pôde ser realizada. Quando a execução de um processo termina, o LRM notifica o fim desta execução ao ASCT. O ASCT então solicita os arquivos de saída da aplicação ao LRM.

O **AR** (*Application Repository*) armazena binários de aplicações submetidas por usuários para execução na Grade. Estes binários são enviados ao AR através do ASCT, de modo a serem posteriormente utilizados pelos LRMs, quando solicitada sua execução.

O **NCC** (*Node Control Center*), não mostrado na figura, permite que os responsáveis pelas máquinas que compartilham seus recursos definam políticas para este compartilhamento. Isto inclui o período em

Figura 3.2: Arquitetura de um aglomerado do InteGrade com suporte à recuperação por retrocesso

que os recursos podem ser compartilhados e a quantidade de recursos disponibilizada¹.

O **LUPA** (*Local Usage Pattern Analyzer*) e **GUPA** (*Global Usage Pattern Analyzer*), não mostrados na figura, realizam a coleta e análise do padrão de utilização das máquinas do aglomerado. O LUPA é executado juntamente com o LRM nos nós provedores de recursos, coletando dados da máquina e analisando seu padrão de uso. Esta informação é disponibilizada ao GRM pelo GUPA, permitindo um escalonamento mais eficiente das aplicações da Grade².

Finalmente, a biblioteca **BSPlib** [56] do InteGrade permite a execução de aplicações paralelas C/C++ do tipo BSP em uma grade InteGrade. Para facilitar o uso de aplicações já existentes, a BSPlib do InteGrade utiliza a mesma API da implementação de Oxford da BSPlib [62].

3.2.2 Recuperação por retrocesso baseada em *checkpointing*

Para implementar o mecanismo de recuperação por retrocesso, adicionamos à arquitetura original do InteGrade: 1) a biblioteca de *checkpointing*, 2) o gerenciador de execuções (*Execution Manager (EM)*), 3) o repositório de *checkpoints* (*Checkpoint Repository (CkpRep)*) e 4) o gerenciador de repositórios de *checkpoints* (*Checkpoint Repository Manager (CRM)*). Na Figura 3.2, apresentamos a arquitetura do InteGrade com os novos componentes. Além disso, diversos outros componentes do InteGrade foram modificados, como o GRM e o LRM, para acomodar as mudanças exigidas pelo nosso trabalho.

Optamos pelo uso de gerenciadores (EM e CRM) centralizados pela facilidade de implementação, incluindo o desenvolvimento de algoritmos mais simples e que requerem menos trocas de mensagens. As versões atuais destes componentes executam no nó de gerenciamento do aglomerado, juntamente com o

¹Até o momento da escrita deste texto, o módulo NCC ainda não havia sido implementado.

²No momento da escrita deste texto, os módulos LUPA e GUPA estavam em fase de implementação.

GRM e o AR do InteGrade. Para evitar a necessidade de máquinas que estão sempre disponíveis para executar os módulos de gerenciamento, poderíamos adicionar tolerância a falhas a esses componentes com uso de replicação e/ou *logging*. O problema de tolerância a falhas nos módulos do InteGrade é o tópico de pesquisa de outros membros do projeto e está fora do escopo desta tese.

Descrevemos abaixo os principais componentes do mecanismo de recuperação por retrocesso baseado em *checkpointing* do InteGrade. Nas Seções 3.3 e 3.4, detalhamos como cada componente foi implementado.

- *Biblioteca de checkpointing*: é responsável por gerar *checkpoints* contendo o estado de um processo e recuperar este estado a partir de um checkpoint. Ela possui funções que permitem à aplicação especificar os dados que serão salvos e os que serão recuperados. Além disso, utilizando um arquivo de configuração, o InteGrade fornece o intervalo de tempo mínimo entre *checkpoints* sucessivos. *Checkpoints* podem ser salvos no sistema de arquivos local ou em repositórios de *checkpoints* remotos. Neste último caso, no momento de armazenar um checkpoint, a biblioteca de *checkpointing* obtém uma lista de repositórios disponíveis para o armazenamento de *checkpoints* com o CRM. Similarmente, para recuperar um checkpoint, a biblioteca de *checkpointing* solicita ao CRM a lista de repositórios contendo o último *checkpoint* salvo durante a execução da aplicação. A biblioteca de *checkpointing* permite que sejam utilizadas diversas estratégias de armazenamento para os checkpoints, como replicação e codificação em múltiplos fragmentos.
- *Repositório de checkpoints*: armazena *checkpoints* e arquivos de saída gerados por processos executando na Grade³. Este repositório provê uma interface remota que permite que dados sejam armazenados e recuperados e se comunica periodicamente com o CRM, o módulo que gerencia os repositórios. Repositórios de *checkpoint* são executados em máquinas provedoras de recursos. Portanto, é possível ter um número de repositórios de *checkpoints* igual ao número de LRMs num aglomerado InteGrade.
- *Gerenciador de execuções (EM)*: mantém uma lista com as aplicações em execução no aglomerado, incluindo a localização de cada processo da aplicação e o estado da requisição de execução. Informações sobre o estado da requisição são fornecidos ao EM pelo GRM, ao escalonar aplicações para execução, ou pelos LRMs, ao iniciarem ou finalizarem a execução de um processo. Estas informações são utilizadas pelo EM para reiniciar aplicações no caso de falha em um dos processos da aplicação. Além disso, estas informações permitem que a execução de uma aplicação seja migrada para outros nós.
- *Gerenciador de Repositórios de Checkpoints (CRM)*: mantém informações sobre os repositórios de *checkpoints* presentes em seu aglomerado, incluindo seu endereço de rede e lista de *checkpoints*

³Os arquivos de saída são parte do estado de um processo.

armazenados. Estes dados são fornecidos à biblioteca de *checkpointing* no momento de armazenar ou recuperar dados de checkpoints. O CRM é também responsável pelo controle dos *checkpoints* armazenados de cada aplicação e determina os arquivos que devem ser mantidos e eliminados. Apenas os *checkpoints* mais recentes de aplicações ainda não terminadas são mantidos.

O InteGrade sempre escalona os processos de uma aplicação paralela em um mesmo aglomerado. Em função disto, desenvolvemos o mecanismo de tolerância a falhas para aplicações cujos processos são executados em um único aglomerado. Para evitar esta restrição seria necessário alterar o EM de modo que EMs de diferentes aglomerados passassem a compartilhar dados sobre os processos sendo executados em cada aglomerado.

3.3 Implementação dos principais componentes

A implementação da infra-estrutura de recuperação por retrocesso é composta por diversos módulos. Nesta seção descrevemos a implementação dos componentes ligados ao gerenciamento de um aglomerado e às máquinas provedoras de recursos.

3.3.1 Módulos de Gerenciamento

Os dois principais módulos de gerenciamento do sistema de recuperação por retrocesso são o Gerenciador de Execuções (EM) e o Gerenciador de Repositórios de *Checkpoints* (CRM). Estes módulos são instanciados no nó de gerenciamento do aglomerado.

Gerenciador de Execuções (EM)

O gerenciador de execuções (EM), implementado em Java, é responsável pelo gerenciamento de requisições de execução de aplicações, incluindo seu reinício em caso de falha de nós. O EM mantém uma tabela interna contendo informações sobre as aplicações escalonadas para execução pelo GRM, incluindo os nós responsáveis pela execução dos processos destas aplicações.

O GRM informa o EM sempre que realiza o escalonamento de uma aplicação, sem fornecer a identificação dos LRMs escolhidos para a execução. Esta identificação é enviada posteriormente pelos próprios LRMs, antes do início da execução dos processos⁴. Quando um processo termina, o LRM informa o EM a respeito do término do processo, assim como o estado (*status*) de término. Deste modo, o EM sempre possui informações atualizadas sobre o estado dos processos das aplicações.

O estado de término de um processo é utilizado pelo EM para determinar se aquele processo deve ser reiniciado ou não. Nos casos em que o processo finaliza normalmente ou em que ocorreu um problema

⁴Caso o número de identificações recebidas pelo EM seja inferior ao número de processos da aplicação paralela, o EM considera que houve falha em um dos LRMs e inicia o protocolo de reinicialização da execução da aplicação, descrito na Seção 3.3.3.

relacionado ao próprio processo, como uma falha de segmentação, o EM não reinicia o processo. Por outro lado, se o processo foi morto por um sinal do sistema operacional, como KILL ou TERM, isto é interpretado como um sinal para realizar a liberação de recursos da máquina. Neste caso, o processo é reiniciado em outro provedor de recursos.

Um outro cenário ocorre quando um LRM se torna inacessível. Isto pode ocorrer por diversos motivos, como partição na rede, problemas na máquina onde o LRM estava executando ou uma falha no próprio LRM. Os LRMs de um aglomerado periodicamente enviam mensagens do tipo *heartbeat* para o GRM. Quando um GRM verifica que um LRM não envia mensagens por um período superior a um limiar, ele envia uma mensagem àquele LRM e, caso não obtenha resposta, marca-o como indisponível. O GRM notifica então o EM, que verifica quais processos estavam sendo executados no LRM que falhou e requisita ao GRM o escalonamento destes processos para execução em outros LRMs.

No caso de aplicações sequenciais e paramétricas, como não existem dependências entre processos, a reinicialização da aplicação é simples, bastando o EM solicitar ao GRM o re-escalonamento do processo que falhou. No caso de processos que fazem parte de aplicações BSP, é preciso considerar também as dependências com os demais processos da aplicação, podendo também ser necessário reiniciar estes processos.

Definimos uma interface CORBA para o EM com métodos que realizam o gerenciamento das execuções de aplicações. Os métodos presentes nesta interface são:

- `reportExecutionScheduled`: É utilizado pelo GRM para notificar que realizou o escalonamento da execução dos processos de uma aplicação e enviou as requisições aos LRMs selecionados. O EM adiciona uma nova entrada à sua tabela de execuções com informações referentes à aplicação escalonada, como número de processos, parâmetros, nós de execução dos processos e arquivos de entrada e saída.
- `reportExecutionStarted`: É utilizado pelos LRMs para notificar o EM que a execução de um processo referente a uma aplicação foi iniciada. Ao receber esta mensagem, o EM adiciona o endereço do LRM às informações da aplicação correspondente àquele processo.
- `reportExecutionFinished`: É utilizado pelos LRMs para notificar que uma execução foi finalizada, enviando o estado de término da execução. O EM executa diferentes ações dependendo do estado de término enviado, como descrito na Seção 3.3.1.
- `notifyLrmFailure`: É utilizado pelo GRM para notificar que um LRM parou de responder. O EM verifica então os processos que estavam sendo executados naquele LRM e chama o método `reportExecutionFinished` para que estes processo sejam reinicializados.
- `registerBspNode`: É utilizado durante a inicialização de aplicações BSP de modo que seus processos possam obter referências ao *processo zero* da aplicação. Cada aplicação BSP possui

um único processo zero, que é responsável por coordenar a execução da aplicação, incluindo sua inicialização e sincronização.

A implementação dos métodos `notifyLrmFailure`, `reportExecutionScheduled` e `reportExecutionStarted` é direta, o único cuidado especial fica por conta da sincronização dos acessos à tabela de execuções. No método `registerBspNode`, para permitir que os processos de uma aplicação BSP possam chamá-lo em qualquer ordem, colocamos em espera quaisquer requisições feitas por processos diferentes do processo zero, utilizando uma variável condicional, até que este se registre. A implementação do método `reportExecutionFinished` é discutida na Seção 3.3.3.

Gerenciador de Repositórios de *Checkpoints* (CRM)

O gerenciador de repositórios de *checkpoints* (CRM), implementado em Java, é responsável pelo gerenciamento de checkpoints de aplicações e de repositórios de checkpoints. O CRM possui duas tabelas principais, que contêm dados sobre: (1) repositórios de *checkpoints* registrados e (2) *checkpoints* de aplicações sendo executadas em seu aglomerado. A interação do CRM com os demais componentes do sistema se dá através de uma interface CORBA, que contém os seguintes métodos:

- `getCheckpointRepositoryList`: É utilizado pela biblioteca de *checkpointing* para obter uma lista de repositórios onde o *checkpoint* pode ser armazenado. Quando a lista de repositórios é solicitada, a biblioteca de *checkpointing* informa qual estratégia de armazenamento está utilizando, replicação ou codificação em múltiplos fragmentos, e os parâmetros desta estratégia, como número de réplicas e fragmentos. A versão atual do CRM escolhe os repositórios aleatoriamente, de modo que, em média, cada repositório armazenará a mesma quantidade de *checkpoints*.
- `getLastCheckpointInfo`: Fornece os dados do último *checkpoint* salvo por uma aplicação, incluindo seu identificador e os locais onde seus fragmentos (ou réplicas) estão localizados.
- `notifyCheckpointStored`: Utilizado por repositórios de *checkpoints* para notificar que um fragmento pertencente a um *checkpoint* foi armazenado. O CRM adiciona às suas tabelas a localização do fragmento armazenado e determina se todos os fragmentos de um *checkpoint* já foram armazenados ou não.
- `registerCheckpointRepository`: É utilizado pelos repositórios de *checkpoints* para se registrar assim que são inicializados. O CRM adiciona os dados do novo repositório à sua lista de repositórios e retorna um identificador único para o repositório.
- `setCheckpointStorageMode`: Configura a estratégia de armazenamento utilizada por uma aplicação. A estratégia de armazenamento determina o número de repositórios fornecidos pelo método `getCheckpointRepositoryList` e o número de fragmentos necessários para determinar um *checkpoint* como armazenado.

- `updateRepositoryStatus`: É chamado periodicamente pelos repositórios para notificar que estão operantes e a quantidade de espaço em disco disponível. Permite ao CRM determinar quais repositórios estão ativos e a quantidade de recursos disponíveis nestes repositórios.
- `getRemovalList`: É chamado pelos repositórios de checkpoints e devolve uma lista de fragmentos de *checkpoints* que podem ser removidos do repositório. Apenas os *checkpoints* mais recentes de aplicações ainda não terminadas são mantidos.

A escolha dos repositórios que armazenarão fragmentos de *checkpoints* é feita de forma aleatória. Deste modo, em média, cada repositório receberá uma quantidade similar de checkpoints. Além disso, antes de selecionar um repositório, o CRM verifica se este possui espaço em disco suficiente para armazenar o fragmento.

Os dados sobre os *checkpoints* gerados por uma aplicação são armazenados no EM, juntamente com os dados de execução da aplicação. Deste modo, o CRM e o EM interagem constantemente. À medida que fragmentos de *checkpoints* são armazenados nos repositórios, estes notificam o CRM. Quando todos os fragmentos de um *checkpoint* global forem armazenados, o CRM marca este *checkpoint* como armazenado.

Os repositórios devem periodicamente enviar ao CRM mensagens para notificar que estão operantes e a quantidade de espaço em disco disponível. Se um repositório passa mais que um determinado tempo, por exemplo, 1 minuto, sem enviar mensagens de notificação, o CRM deixa de selecioná-lo para a armazenagem de checkpoint. Se este tempo for maior que um segundo limiar, por exemplo, 1 hora, o CRM remove de suas tabelas todas as informações relativas àquele repositório.

O CRM mantém uma lista de remoção para cada repositório de checkpoints, contendo a lista de *checkpoints* que devem ser removidos. Para determinar estes checkpoints, o CRM interage com o EM de seu aglomerado. Sempre que uma aplicação finaliza sua execução, o EM notifica o CRM, que coloca todos os *checkpoints* daquela execução na lista de remoção. Além disso, o CRM permite manter um número máximo de *checkpoints* por execução, de modo que à medida que novos *checkpoints* são gerados, outros são removidos. Utilizando um protocolo de *checkpointing* coordenado, basta manter o último *checkpoint* global gerado pela aplicação.

Mas podemos melhorar as propriedades de tolerância a falhas do mecanismo de armazenamento mantendo um número maior de *checkpoints* por aplicação. Durante o reinício de uma aplicação, caso não seja possível recuperar o último *checkpoint* global desta aplicação, por exemplo porque parte dos repositórios contendo fragmentos do *checkpoint* ficaram inacessíveis, pode-se utilizar um *checkpoint* anterior.

3.3.2 Componentes para provedores de recursos

Os dois componentes envolvidos no processo de recuperação por retrocesso que executam nos nós provedores de recurso são o Gerenciador de Recursos Locais (LRM) e o Repositório de Checkpoints (CkpRep).

Gerenciador de Recursos Locais (LRM)

O LRM, implementado em C++, possui a responsabilidade de gerenciar a execução de processos em uma máquina, incluindo sua inicialização e monitoramento e a coleta de resultados. Quando a execução de um processo inicia ou termina, o LRM notifica o gerenciador de execuções do seu aglomerado. No caso do término da execução, o LRM envia também o estado de término do processo. Esse estado identifica se a aplicação finalizou normalmente, se ela foi morta por um sinal externo ou se o próprio LRM decidiu terminar com a execução. O LRM pode terminar uma execução, por exemplo, porque o dono da máquina requisitou os seus recursos de volta ou porque o usuário que submeteu a aplicação decidiu cancelar sua execução.

O LRM possui um método `void kill(string processId, boolean restart)` em sua interface CORBA que permite finalizar um processo identificado por `processId` sendo executado na máquina. Dependendo do valor do parâmetro `restart`, o LRM solicita ou não a reinicialização deste processo para o gerenciador de execuções.

Repositórios de *Checkpoints* (CkpRep)

O *Repositório de Checkpoints*, implementado em C++, funciona como um servidor que espera por requisições remotas para armazenamento e recuperação de dados. A comunicação com clientes é realizada utilizando soquetes TCP/IP, obtendo assim um melhor desempenho na transmissão de dados. Para facilitar o uso do repositório, um *broker*, contendo funções para envio e recebimento de dados por parte do cliente, é fornecido. Este *broker* é utilizado pela biblioteca de *checkpointing* para armazenar e recuperar *checkpoints* de um processo. Já a comunicação do repositório com o CRM é realizada através da interface CORBA do CRM, utilizando o ORB OiL [128], escrito em Lua.

O repositório armazena dados recebidos no disco local da máquina onde é executado. O local onde os arquivos serão colocados é especificado como parâmetro para o repositório e deve apontar para um diretório que represente um sistema de arquivos local.

3.3.3 Reinicialização de processos e aplicações

Uma aplicação pode precisar ser reinicializada por diversos motivos, como falha na máquina onde um processo da aplicação era executado ou porque o dono da máquina solicitou seus recursos de volta. No primeiro caso, o GRM detecta a falha na máquina quando um LRM pára de responder. Neste caso,

além de remover este LRM da lista de LRMs disponíveis, o GRM notifica o EM, de modo que este possa realizar o processo de reinicialização de todas as aplicações que possuam processos sendo executados naquela máquina. No caso em que o dono da máquina solicita os recursos de volta, o LRM inicialmente envia um sinal do tipo KILL a todos os processos executando naquela máquina e notifica o EM sobre o término prematuro destes processos.

Em ambos os casos, o método `reportExecutionFinished` do EM, responsável pela reinicialização de processos e aplicações, é chamado. Este método procede de maneiras distintas, dependendo do tipo de aplicação que terminou e do seu *status* de término. Caso o processo tenha sido finalizado para liberar recursos da máquina ou porque a máquina ficou inacessível, o EM inicia o procedimento de reinicialização da aplicação. Caso contrário, o EM marca o processo como tendo terminado por um erro de execução e reporta o erro ao ASCT.

Com relação ao tipo de aplicação que terminou, se a aplicação for sequencial ou paramétrica, como não existem dependências entre os processos da aplicação, basta o EM requisitar ao GRM o escalonamento do processo que terminou. Para aplicações BSP, além de solicitar ao GRM escalonamento do processo que falhou, o EM precisa requisitar aos LRMs a reinicialização de todos os demais processos da aplicação. Isto é feito através de um método presente nos LRMs denominado `restartProcess`, que reinicia um processo. As solicitações de reinicialização dos processos da aplicação aos LRMs são realizadas em paralelo com o escalonamento do processo que falhou.

Quando um processo é reinicializado, a biblioteca de *checkpointing* solicita ao CRM de seu aglomerado o identificador do último *checkpoint* gerado e os endereços dos repositórios onde este *checkpoint* está armazenado. No caso de aplicações sequenciais ou paramétricas, a biblioteca de *checkpointing* obtém o arquivo de *checkpoint* dos repositórios devolvidos pelo CRM, lê o conteúdo destes arquivos e reinicializa o processo a partir dos dados contidos no *checkpoint*. No caso de aplicações paralelas BSP, é necessário ainda sincronizar os processos da aplicação e fornecer a cada processo os endereços dos demais processos da aplicação. Para tal, a BSPLib ligada a cada processo chama o método `registerBspNode` do EM, que devolve ao processo o endereço do processo zero. A BSPLib de cada processo chama então o processo zero, que coordena a inicialização da aplicação BSP e fornece a cada processo os endereços dos demais processos da aplicação.

É necessário realizar a sincronização entre diferentes reinicializações que podem ocorrer simultaneamente. No caso de requisições de execução distintas, basta sincronizar os acessos às tabelas do EM. Mas no caso de reinicializações relativas a uma mesma requisição de execução, são necessários cuidados extras. Em particular, é possível que durante o processo de reinicialização de uma aplicação BSP ocorra uma falha em outro processo desta mesma aplicação. Neste caso, o EM realiza um novo processo de reinicialização da aplicação. Além disso, é possível que o LRM ou a máquina na qual algum dos processos de uma aplicação está sendo executado também falhe durante a reinicialização. O tratamento neste caso é similar ao caso de falha de um processo, uma vez que o método `NotifyLrmFailure` chama o método `reportExecutionFinished` do EM para cada processo que estava sendo executado na

máquina que falhou.

Figura 3.3: Diagrama de sequência da reinicialização de uma aplicação BSP composta por 2 processos.

A Figura 3.3 mostra um diagrama de seqüências para a liberação de recursos em uma máquina que está executando um processo de uma aplicação BSP composta por 2 processos. O LRM envia um sinal do tipo KILL ao processo 0 e notifica o EM sobre o término prematuro daquele processo. O EM verifica que aquele processo faz parte de uma aplicação BSP composta por 2 processos e simultaneamente inicia o processo de reinicialização dos 2 processos: processo 0 e processo 1.

No caso do processo 0, o EM solicita ao GRM o reescalonamento daquele processo. O GRM seleciona o LRM 3 e lhe envia uma requisição para execução do processo 0. LRM 3 inicia então a execução do processo 0. Já no caso do processo 1, o EM solicita ao LRM 2, presente na máquina onde o processo 1 é executado que aquele processo seja reiniciado. LRM envia então um sinal KILL ao processo 1 e lança uma nova execução deste processo.

Ambos os processos iniciam sua execução e a biblioteca de *checkpointing* (ckpLib) ligada a cada

processo solicita ao CRM o endereço do repositório onde o último checkpoint gerado está armazenado. A ckpLib de cada processo obtém este *checkpoint* a partir do repositório devolvido e utiliza-o para reiniciar o processo a partir do estado contido neste *checkpoint*. A BSPLib requisita então ao EM o endereço do processo 0, que será então utilizado para inicializar a aplicação BSP.

3.4 Pré-compilador e biblioteca de *checkpointing*

Os dois componentes principais para a geração de *checkpoints* a partir do estado de uma aplicação são o pré-compilador e a biblioteca de *checkpointing*. Nesta seção descrevemos a implementação destes dois componentes. Disponibilizamos o código fonte destes componentes na página do projeto na Internet (<http://www.integrate.org.br/portal/software/checkpointing>).

3.4.1 Pré-compilador

A implementação do pré-compilador é baseada na ferramenta de código-aberto OpenC++ [32]. Esta ferramenta foi concebida para realizar tarefas de meta-computação através do uso de meta-objetos que atuam em tempo de compilação. A grande vantagem de se utilizar meta-objetos em tempo de compilação é a eficiência, uma vez que não existe uma sobrecarga em tempo de execução. Em particular, através da meta-programação esta ferramenta permite que extensões para C++ sejam utilizadas.

O OpenC++ recebe como entrada um código C++ pré-processado⁵ e realiza a análise léxica e sintática do código, construindo uma árvore sintática que representa o código fonte. Para cada nó desta árvore, métodos de meta-objetos específicos para aquele tipo de nó são chamados, podendo realizar modificações no código fonte da aplicação. Existem métodos que são chamados quando um membro de uma classe é modificado, quando um método da classe é declarado, quando um método é chamado, entre outros. Após todos os nós serem visitados, o código C++ modificado é reconstruído a partir desta árvore sintática.

A programação utilizando o protocolo de meta-objetos consiste em definir meta-objetos descendentes de uma classe `Class`, que possui diversos métodos virtuais correspondentes a ações que podem ser realizadas sobre o meta-objeto. Por exemplo, se o objetivo é modificar as chamadas de métodos, o meta-objeto deve sobrescrever o método `TranslateFunctionCall`, que é chamado sempre que o código realiza uma chamada de método de uma classe.

Nosso objetivo inicial era o de utilizar estes meta-objetos para modificar o código fonte das aplicações. No entanto, o protocolo para meta-objetos disponibilizado não era genérico o suficiente para nossas tarefas. Por exemplo, precisávamos determinar, dentro de uma função, as posições relativas entre declarações de variáveis e chamadas de métodos.

⁵O pré-processador de um compilador realiza um processamento inicial no código fonte de uma aplicação, que inclui a expansão de macros, inclusão de macros, remoção de comentários, compilação condicional, etc.

Optamos por abandonar o protocolo de meta-objetos do OpenC++ e modificar seu código fonte diretamente. Internamente, o OpenC++ invoca métodos específicos para cada tipo de nó da árvore sintática sendo analisado; por exemplo, existem métodos que são chamados quando uma variável local é declarada, quando um método é invocado ou quando uma classe é definida. Estes métodos realizam o processamento do código e são a base do protocolo de meta-objetos do OpenC++. Estendemos estes métodos de processamento para realizar a tarefa extra de instrumentar o código fonte sendo analisado. A vantagem de trabalhar internamente no OpenC++ é que temos acesso a todas suas classes internas, incluindo as que representam o código sendo analisado, de modo que possuímos informação completa sobre este código.

Modificamos também o protocolo de compilação do OpenC++ para analisar a árvore sintática duas vezes ao invés de apenas uma. Isto porque, antes de modificar o código fonte da aplicação, é necessário obter diversas informações sobre este código. Por exemplo, como veremos na Seção 3.4.2, apenas parte das funções do código fonte precisam ser modificadas. Além disso, apenas as variáveis locais que estão “visíveis” no ponto de criação de um *checkpoint* precisam ser salvas. Estas informações são obtidas na primeira análise, enquanto na segunda análise o código da aplicação é modificado.

O pré-compilador é utilizado para inserir código de *checkpointing* e para modificar chamadas da aplicação à API do BSP, de modo a permitir a comunicação entre processos executando em máquinas de diferentes arquiteturas. Ambas as tarefas são realizadas simultaneamente pelo pré-compilador.

A versão atual do pré-compilador instrumenta aplicações em C, com algumas extensões provenientes do C++. Estas extensões incluem o suporte à declaração de variáveis em qualquer ponto do corpo de um método, e não apenas em seu início. Futuramente iremos estender o pré-compilador para trabalhar com aplicações C++ orientadas a objetos. Será necessário tratar das diversas características de C++, como herança, polimorfismo, templates, contêineres da STL e referências C++.

Finalmente, as técnicas aqui descritas para a instrumentação do código fonte de uma aplicação podem também ser utilizadas em outras linguagens procedurais, como Java e Fortran. Mas é necessário implementar um pré-compilador específico para cada linguagem de programação.

3.4.2 Instrumentação do código de uma aplicação

Como vimos na Seção 2.2.3, o estado de um processo é composto por seu espaço de memória, pelos valores presentes em registradores do processador e pela parte do estado do sistema operacional que armazena informações sobre aquele processo. A seguir, detalhamos nossa implementação do mecanismo de *checkpointing* portátil.

Estado da pilha de execução

Para obter o estado da pilha de execução de um processo em um determinado momento de sua execução, de modo que este estado seja posteriormente recuperável, basta armazenarmos a lista das

funções ativas e os valores de parâmetros e variáveis locais destas funções naquele momento. Para recuperar a pilha de execução, basta chamar as funções presentes nesta lista e inicializar os parâmetros e variáveis locais destas funções com os valores armazenados.

A biblioteca de *checkpointing* fornece uma *pilha de checkpointing* onde endereços de variáveis locais e de parâmetros de funções são empilhados, permitindo a obtenção do conteúdo destas variáveis locais e parâmetros. O pré-compilador modifica o código fonte da aplicação de modo que os endereços de variáveis locais e parâmetros são adicionados a esta pilha assim que são declarados. No momento de gerar um novo *checkpoint*, a biblioteca de *checkpointing* obtém os dados que serão salvos a partir dos endereços contidos na pilha de *checkpointing*. Uma vez que estes dados são lidos a partir de seus endereços somente no momento de criação do *checkpoint*, é garantido que estes dados estarão atualizados.

A lista de funções na pilha é armazenada utilizando a mesma *pilha de checkpointing*. O pré-compilador insere no início do corpo das funções da aplicação uma nova variável local, denominada `lastFunctionCalled`, que é inicializada com o valor -1 e é incrementada antes da chamada de cada função. No momento em que o *checkpoint* é gerado, o valor de `lastFunctionCalled` para cada função é armazenado.

O *contador de programa* mantém um ponteiro para a próxima instrução que será executada pelo processador. Para capturar qual a próxima instrução a executar, *checkpoints* são gerados apenas em pontos pré-determinados da execução, quando a aplicação chama a função `checkpoint_candidate`, definido pela biblioteca de *checkpointing*. As chamadas a esta função são inseridas pelo desenvolvedor da aplicação nos pontos onde este deseja que *checkpoints* possam ser criados. O armazenamento da chamada desta função na pilha de *checkpointing* é similar ao das demais funções, com a diferença que quando a execução atinge esta função, a biblioteca de *checkpointing* gera um novo *checkpoint*⁶.

Durante a reinicialização da aplicação, os parâmetros e variáveis locais da função `main`, incluindo `lastFunctionCalled`, são declarados e seus valores lidos do *checkpoint*, de modo que o estado da função `main` quando o *checkpoint* foi gerado é recuperado. A execução da aplicação pula então diretamente para a chamada da função correspondente ao valor de `lastFunctionCalled`, ignorando o código da função que foi pulado. Assim que esta função é chamada, o mesmo procedimento de recuperação de valores de parâmetros e variáveis locais é realizado, e a execução pula para a chamada da função correspondente ao valor de `lastFunctionCalled`. Este procedimento é repetido, até a execução chegar à função `checkpoint_candidate`, que corresponde ao exato ponto da execução onde o *checkpoint* foi gerado.

O algoritmo abaixo ilustra o procedimento descrito. As linhas descritas por (. . .) equivalem a uma sequência de comandos realizada pela função.

```
1  /**
2  * Função original
```

⁶A biblioteca de *checkpointing* permite definir um intervalo mínimo entre *checkpoints*, de modo que um novo *checkpoint* só é gerado se o intervalo de tempo desde a criação do último *checkpoint* for maior que este intervalo mínimo.

```

3  */
4  int functionA (int parameter) {
5      int localInt = 0;
6      (...) // Outros comandos
7      double localDouble = 0;
8      (...) // Outros comandos
9      function0 ();
10     (...) // Outros comandos
11     function1 ();
12     (...) // Outros comandos
13     checkpoint_candidate ();
14     (...) // Outros comandos
15     return 0;
16 }

```

Apresentamos agora a mesma função após ser modificada pelo pré-compilador. Neste exemplo é possível visualizar as principais modificações realizadas pelo pré-compilador.

```

1  /**
2   * Após a instrumentação utilizando o pré-compilador
3   */
4  int functionA (int parameter) {
5      int lastFunctionCalled = -1;
6      int localInt = 0 ;
7      ckp_push_data(&lastFunctionCalled , sizeof(int), CKPT_INT);
8      ckp_push_data(&parameter , sizeof(int), CKPT_INT);
9      ckp_push_data(&localInt , sizeof(int), CKPT_INT);
10     if (ckp_recovering==1) {
11         ckp_get_data(&lastFunctionCalled , sizeof(int));
12         ckp_get_data(&parameter , sizeof(int));
13         ckp_get_data(&localInt , sizeof(int));
14         goto var0;
15     }
16     (...) // Outros comandos
17 var0:
18     double localDouble = 0;
19     ckp_push_data(&localDouble , sizeof(double), CKPT_DOUBLE);
20     if (ckp_recovering==1) {
21         ckp_get_data(&localDouble , sizeof(double), CKPT_DOUBLE);
22         if(lastFunctionCalled == 0)
23             goto ckp0;
24         if(lastFunctionCalled == 1)
25             goto ckp1;
26     }
27     (...) // Outros comandos
28 ckp0:

```



```
29  lastFunctionCalled = 0;
30  function0 ( ) ;
31  (...) // Outros comandos
32  function1 ( ) ;
33  (...) // Outros comandos
34  ckp1:
35  lastFunctionCalled = 2;
36  int createCheckpoint = checkpoint_candidate() ;
37  if (createCheckpoint == 1)
38      ckp_save_stack_data();
39  (...) // Outros comandos
40  ckp_npop_data(3);
41  return 0;
42 }
```

Neste código, as funções `ckp_push_data` e `ckp_get_data` permitem que a aplicação forneça os dados a serem salvos e os recupere posteriormente. Estas funções recebem, como parâmetro, o endereço da variável, o seu tamanho e o seu tipo. No momento em que o *checkpoint* é gerado, o tipo é utilizado apenas no caso de variáveis que são ponteiros. Durante a recuperação, o tipo é utilizado para a conversão de representação dos dados para a arquitetura de destino à medida que estes são lidos. É interessante notar que as conversões são realizadas apenas no momento da recuperação dos dados. Isto é importante, uma vez que cada execução de um processo normalmente gera muitos *checkpoints*, utilizando-os para reinicialização um número muito menor de vezes. Além disso, as conversões são realizadas apenas quando as arquiteturas onde o *checkpoint* foi gerado e está sendo recuperado são diferentes.

A variável `ckp_recovering` é uma variável global, declarada e inicializada pela biblioteca de *checkpointing* de cada processo, à qual é atribuída o valor 1 se o processo está sendo reinicializado e 0 caso contrário. Como podemos ver nas linhas 10 e 20, o código de recuperação dos valores das variáveis locais só é executado quando `ckp_recovering` possui valor 1.

É possível ainda visualizar que, durante a reinicialização, apenas comandos do código original relativos à declaração de variáveis e chamadas de algumas funções são executados, como podemos ver ao analisarmos os comandos `goto`, nas linhas 14, 23 e 25, e seus respectivos destinos. Os demais comandos são ignorados, incluindo as atribuições de valores a variáveis, uma vez que estes valores são obtidos a partir do *checkpoint*. Além disso, vemos que as funções `function0`, na linha 30, e `function1`, na linha 32, são tratadas de modo diferente pelo pré-compilador. Como veremos a seguir, apenas um subconjunto das funções são relevantes à geração de *checkpoints*. No exemplo acima, `function1` não é relevante, sendo portanto ignorada pelo pré-compilador.

Podemos verificar a função `checkpoint_candidate`, na linha 36, que devolve 1 se um novo *checkpoint* deve ser gerado e 0 caso contrário. Para gerar o *checkpoint*, a aplicação chama o método `ckp_save_stack_data`, que gera o *checkpoint* a partir dos valores contidos na pilha de *checkpointing*.

Finalmente, a chamada de função `ckp_npop_data(3)`, na linha 40, indica que 3 endereços devem ser retirados da pilha de *checkpointing*. Isto ocorre porque, no momento em que o comando `return` é executado, estas variáveis deixam de ser “visíveis” e, portanto, não devem ser salvas. A função `ckp_npop_data` é chamada sempre que alguma variável deixa de ser “visível”, por exemplo, quando a execução sai de um bloco, e não apenas no final de funções. O pré-compilador precisa determinar, para cada variável, em quais trechos de código esta variável é “visível”.

Conjunto de funções que precisam ser modificadas.

Apenas um subconjunto das funções de uma aplicação precisam ser modificadas. Este subconjunto inclui todas as funções que podem estar na pilha de execução durante a criação de um checkpoint. Este conjunto, que denominaremos ϕ neste trabalho, é definido da seguinte forma:

Definição 3.1 *Seja ϕ o conjunto das funções que precisam ser modificadas. Uma função $f \in \phi$ se, e somente se:*

- (i) *f é responsável por gerar um checkpoint⁷;*
- (ii) *f chama uma função $g \in \phi$.*

Para determinar quais funções precisam ser salvas, o pré-compilador inicialmente determina as funções que geram checkpoints, para então, iterativamente, determinar todas as demais funções que pertencem a ϕ .

Ponteiros

Ponteiros podem referenciar endereços pertencentes à pilha de execução ou à memória dinâmica (*heap memory*). Estes endereços referenciados são específicos a uma execução particular da aplicação e normalmente possuem uma representação diferente para cada arquitetura.

A biblioteca de *checkpointing* converte os endereços de memória referenciados por ponteiros em *offsets* no arquivo de checkpoint. Durante a recuperação, estes *offsets* são então convertidos para endereços de memória alocados na arquitetura de destino. Esta estratégia é aplicada tanto para ponteiros que referenciam memória alocada dinamicamente como para ponteiros para endereços na pilha de execução.

Para permitir esta conversão de endereços em *offsets*, a biblioteca de *checkpointing* mantém uma tabela que guarda os endereços dos blocos de memória alocados, bem como seus tamanhos e posições no *buffer* de *checkpointing*. O pré-compilador substitui chamadas de sistema relativas à alocação de memória – `malloc`, `realloc` e `free` – no código da aplicação por funções equivalentes da biblioteca de *checkpointing*. Estas funções atualizam a tabela de endereços de memória alocados antes de repassarem

⁷Dizemos que uma função é responsável por gerar um *checkpoint* se esta chama uma função da biblioteca de *checkpointing* que inicia o processo de armazenamento do estado da aplicação, no nosso caso, `checkpoint_candidate()`.

a chamada ao sistema. O último campo da tabela, com a posição dos blocos de memória no *buffer* de *checkpointing*, é utilizado durante a criação dos checkpoints. Na Seção 3.4.3, descreveremos como é realizado o processo de gerar um checkpoint.

Estruturas

A representação de estruturas C na memória varia de acordo com a arquitetura e compilador utilizados. O alinhamento de dados é realizado tanto em função de requisitos arquiteturais como para melhorar o desempenho. Para evitar ter que manipular o alinhamento de dados, optamos pela abordagem de empilhar o endereço de cada membro da estrutura na pilha de *checkpointing*. Deste modo, os espaços de alinhamento são automaticamente eliminados, com a vantagem extra de reduzir o tamanho do checkpoint, uma vez que os bytes que ocupam os espaços vazios entre os membros da estrutura não são armazenados.

Para cada estrutura presente no código fonte, o pré-compilador cria duas funções, uma para salvar e a outra para recuperar os dados da estrutura. A função que salva os dados empilha na pilha de *checkpointing* todos os membros da estrutura, enquanto a função que recupera os dados obtêm os valores destes membros individualmente a partir do checkpoint. Abaixo vemos um exemplo de uma estrutura e as respectivas funções, adicionadas pelo pré-compilador.

```
1 struct SimpleNode{
2     int value;
3     SimpleNode *next;
4 };
5
6 void ckp_save_SimpleNode(void *data){
7     SimpleNode *str = (SimpleNode *)data;
8     ckp_push_data(&(str->value), sizeof(int), CKPT_INT, 0);
9     ckp_push_data(&(str->next), sizeof(SimpleNode),
10                 CKPT_STRUCT_POINTER, &ckp_save_SimpleNode);
11 }
12 void ckp_restore_SimpleNode(void *data){
13     SimpleNode *str = (SimpleNode *)data;
14     ckp_get_data(&(str->value), sizeof(int), CKPT_INT, 0);
15     ckp_get_data(&(str->next), sizeof(SimpleNode),
16                 CKPT_STRUCT_POINTER, &ckp_restore_SimpleNode);
17 }
```

O ponto interessante neste exemplo é o membro *next*, na linha 3, da estrutura *SimpleNode*, que aponta para outra estrutura *SimpleNode*. Ao chamar, na linha 9, *ckp_push_data* para empilhar o membro *next*, um parâmetro adicional, contendo o endereço da função que salva esta estrutura, é passado à função. A biblioteca de *checkpointing* utiliza este endereço para chamar recursivamente a função *ckp_save_SimpleNode*, que salva a estrutura apontada por *next*. O parâmetro adicional

em `ckp_get_data`, na linha 15, funciona de modo similar.

Similarmente, quando uma estrutura é declarada no corpo de um método, o pré-compilador adiciona os comandos `ckp_push_data` e `ckp_get_data` passando o endereço da função que recupera ou salva os dados da estrutura. No caso em que a estrutura é alocada na pilha de execução, a biblioteca de *checkpointing* chama esta função, passada como parâmetro, imediatamente. Já no caso onde é empilhado um ponteiro para estrutura, o pré-compilador espera o momento em que o *checkpoint* é gerado para chamar a função que salva a estrutura, uma vez que os endereços referenciados por ponteiros podem mudar durante a execução da aplicação.

3.4.3 Criação de *checkpoints* e recuperação do estado da aplicação

Veremos agora o modo através do qual os dados de um processo são armazenados em um *checkpoint* e posteriormente recuperados numa arquitetura possivelmente distinta da original.

Criação do *checkpoint*

Um novo *checkpoint* é gerado sempre que a aplicação chama o método `ckp_save_stack_data`. Como já vimos, a biblioteca de *checkpointing* precisa substituir os endereços de memória por *offsets*. O processo de gerar o *checkpoint* é realizado em três fases:

1. Na primeira fase, é calculado o tamanho necessário para o *buffer* que irá armazenar o *checkpoint* temporariamente. Neste processo, endereços de blocos de memória alocados dinamicamente são inseridos na pilha de *checkpointing*. Definimos também uma pilha de ponteiros, onde empilhamos os índices da pilha de *checkpointing* onde estão localizados os endereços dos ponteiros e os dados referenciados por estes ponteiros. Nesta fase, ponteiros com múltiplos níveis de indireção são também resolvidos, incluindo matrizes multi-dimensionais alocadas dinamicamente. Neste caso, todos os ponteiros presentes na matriz são empilhados na pilha de ponteiros. Finalmente, no caso de ponteiros para estruturas, a função que armazena os membros da estrutura é chamada. Esta função, por sua vez, empilha novos itens na pilha de *checkpointing*, que serão tratados ainda nesta fase à medida em que a pilha for percorrida.
2. Na segunda fase, tipos de dados primitivos e blocos de memória alocados dinamicamente são copiados no *buffer* de *checkpoint* e suas localizações são inseridas em uma tabela, que contém as suas posições no *buffer*.
3. Na terceira fase, os endereços de memória contidos nos ponteiros são substituídos por *offsets* contidos na tabela de posições no *buffer*. Devemos notar que, ao utilizar estes *offsets*, referências para endereços repetidos e referências circulares são tratadas automaticamente, com os ponteiros referenciando a mesma posição no *buffer*.

Estas três fases requerem $O(s + p)$ passos, onde s é o número de elementos na pilha de *checkpointing* antes da geração do *checkpoint* e p é o número de elementos na pilha de ponteiros no final da primeira fase. A biblioteca de *checkpointing* permite salvar os *checkpoints* diretamente no sistema de arquivos ou em repositórios remotos de checkpoints.

Recuperação do estado contido em um checkpoint

A recuperação dos dados de um *checkpoint* é realizada pela classe `CkpRestoreData`. Inicialmente os dados do *checkpoint* são lidos para um *buffer* temporário. À medida em que a aplicação realiza chamadas ao método `ckp_get_data`, os dados são copiados do *buffer* para endereços na pilha de execução e para a memória dinâmica da aplicação, de modo a recuperar o estado armazenado no *checkpoint*. Para determinar o próximo dado que será lido do *buffer*, a classe `CkpRestoreData` mantém um ponteiro para a posição corrente no *buffer*.

O parâmetro contendo o tipo do dado a ser lido, enviado na chamada a `ckp_get_data`, determina como este dado será tratado. Se for um tipo primitivo, os bytes referentes àquele tipo serão copiados no endereço fornecido na chamada a `ckp_get_data`, realizando, quando a arquitetura onde o *checkpoint* foi gerado for diferente, a conversão apropriada na representação dos dados. No caso de estruturas, a biblioteca chama as funções fornecidas na chamada a `ckp_get_data`, de modo a reconstruir a estrutura, como descrito na Seção 3.4.2. Finalmente, no caso de ponteiros, o procedimento depende do tipo de ponteiro armazenado:

- Para *ponteiros simples* (com um nível de indireção) para endereços de *memória alocada dinamicamente*, é preciso determinar primeiro se o bloco de memória referenciado já foi lido do *buffer*.
 - Em caso negativo, o tamanho do bloco é lido a partir do *buffer* e um novo bloco de memória alocado dinamicamente. Os dados do *buffer* são então copiados para o bloco de memória recém alocado. Se a arquitetura onde o *checkpoint* está sendo restaurado for igual à que ele foi salvo, os dados são copiados diretamente para o endereço fornecido. Caso contrário, os dados do *buffer* são copiados em blocos do tamanho do tipo de dado apontado pelo ponteiro fornecido, sendo convertidos à medida em que são copiados. No final da cópia, o endereço do bloco de memória alocado é adicionado a uma tabela `dataChunksRead`, indexada pelo *offset* do bloco que acabou de ser copiado do *buffer*.
 - Em caso afirmativo, é atribuído ao ponteiro fornecido o endereço de memória contido na posição *offset* da tabela `dataChunksRead`.
- No caso de *ponteiros com múltiplos níveis de indireção*, o procedimento é similar ao caso de ponteiros simples. Mas, ao realizar a cópia dos dados para o bloco de memória alocado, é necessário agora substituir os *offsets* contidos no bloco de memória do *buffer* por endereços de memória reais,

no espaço da aplicação. Para cada *offset* contido no bloco de memória, é realizada uma chamada a `ckp_get_data`, reduzindo o nível de indireção do ponteiro. Este processo é repetido recursivamente até o ponteiro se transformar em um ponteiro simples.

- No caso de ponteiros para endereços da *pilha de execução* o procedimento é diferente. Os endereços destes ponteiros são armazenados em uma tabela `stackDataPointersList`, junto com os *offsets* lidos do *buffer* temporário. À medida que os dados são lidos do *buffer* e copiados para a pilha de execução, os *offsets* e respectivos endereços na pilha de execução são adicionados a uma tabela `stackDataAddresses`, similar à tabela `dataChunksRead`. No final do processo de recuperação do estado da aplicação, após todos os dados do *buffer* serem lidos, a tabela `stackDataPointersList` é percorrida, atribuindo aos endereços de memória lá contidos os valores dos endereços de `stackDataAddresses`.

Existem ainda outros casos que são tratados pela biblioteca de *checkpointing*, como vetores alocados diretamente na pilha de execução e ponteiros para estrutura, e que não descreveremos neste texto⁸. O procedimento para recuperar os dados nestes casos é similar aos demais, diferindo apenas em detalhes de implementação.

3.4.4 Transformação de dados

Existem duas possibilidades para realizar transformações de dados entre n arquiteturas diferentes. Podemos definir conversores para cada par de arquiteturas, o que resulta em $n^2 - n$ conversores, ou definir uma representação intermediária, como em XDR [116], de modo que cada arquitetura tenha um par de conversores (ida e volta), resultando em $2n$ conversores.

A primeira abordagem, conversão direta, realiza uma conversão mais rápida, uma vez que não há a passagem por uma representação intermediária. Além disso, sempre que usamos uma representação intermediária, ao definir esta representação é necessário encontrar um equilíbrio entre precisão dos dados e uso de memória. A desvantagem de realizar a conversão direta é que o número de conversores aumenta quadraticamente com o número de arquiteturas. Entretanto, é possível utilizar uma abordagem híbrida para o caso de não existirem conversores para todos os pares de arquiteturas, bastando que utilizemos a representação de uma das arquiteturas como representação intermediária.

Observando estas vantagens e desvantagens, optamos pela utilização da conversão direta. Implementamos conversores para as arquiteturas normalmente utilizadas em estações de trabalho, que são x86, x86_64 e ppc, de modo que o número de conversores ficou relativamente baixo. Definimos uma classe para cada tipo de conversão, contendo um método para cada tipo de dado primitivo. Endereços referenciados por ponteiros não precisam ser convertidos, uma vez que, em aplicações BSP, são transmi-

⁸Leitores interessados podem analisar os procedimentos diretamente no código fonte da biblioteca de *checkpointing*, disponível na página do projeto InteGrade [123].

tidos sempre blocos de memória, enquanto no processo de *checkpointing*, os endereços são convertidos em posições relativas.

As representações de dados nas arquiteturas x86 e x86_64 são bastante similares. Ambas utilizam a ordenação *little endian*, diferindo apenas nas representações dos tipos `long` e `long double`. Já a arquitetura PowerPC G4 utiliza a ordenação *big endian*, de modo que é necessário realizar conversões para todos os tipos de dados, com exceção daqueles que utilizam apenas um byte, no caso, os tipos `char` e `unsigned char`. Estas três arquiteturas diferem ainda com relação ao alinhamento de dados e à representação de endereços de memória, mas em função das características de nossa abordagem, estas diferenças não interferem na portabilidade do checkpoint, não sendo necessário escrever código para realizar conversões.

3.4.5 Armazenamento e recuperação de checkpoints

Para permitir o armazenamento e recuperação de *checkpoints* em repositórios remotos utilizando diferentes estratégias de armazenamento, incorporamos à biblioteca de *checkpointing* um *broker* que se comunica com o gerenciador de repositórios de checkpoints (CRM) e com os repositórios de checkpoints.

Durante o armazenamento de um checkpoint, o *broker* é responsável por conectar-se ao CRM, obter informações sobre os repositórios disponíveis, codificar os *checkpoints* e transferi-los para os repositórios. Do mesmo modo, para recuperar um checkpoint, o *broker* realiza todo processo de obtenção da localização dos fragmentos do checkpoint, recuperação do conteúdo destes fragmentos e reconstrução do checkpoint. A biblioteca de *checkpointing* acessa o armazenamento remoto utilizando dois métodos, `saveCheckpointData` e `recoverCheckpointData`, passando como parâmetros um identificador do processo e do *checkpoint* a ser salvo ou recuperado. A estratégia de armazenamento é definida através de um arquivo de configuração, “ckp.conf”, que é colocado no diretório de execução do processo.

Para melhorar o desempenho durante a reinicialização de uma aplicação, a biblioteca de *checkpointing* pode salvar, além da cópia remota, uma cópia local do checkpoint. Deste modo, é possível aumentar a disponibilidade dos *checkpoints* sem aumentar a quantidade de dados transferidos pela rede. Além disso, durante a reinicialização de uma aplicação paralela, os processos já terão os seus últimos *checkpoints* disponíveis localmente. Este armazenamento local do *checkpoint* só faz sentido para o caso de aplicações paralelas. No caso de uma aplicação de apenas um processo, este normalmente precisa ser reiniciado em outra máquina, tornando esta cópia local inacessível.

Estratégias de armazenamento

O *broker* pode utilizar diversas estratégias para codificação e armazenamento de checkpoints, discutidas na Seção 2.4.1. A versão atual do *broker* possui suporte ao uso de replicação, de paridade e do algoritmo de dispersão de informação (IDA).

Quando é utilizada replicação para armazenar checkpoints, o *broker* armazena r réplicas do *checkpoint* em repositórios distintos. Para realizar a recuperação do checkpoint, basta que o *broker* obtenha uma destas réplicas.

O *broker* implementa também o algoritmo de paridade descrito na Seção 2.4.1. Ele quebra o *checkpoint* em m fragmentos e gera um fragmento extra contendo a paridade dos demais fragmentos utilizando operações ou-exclusivo. Em Java e C++, a operação de ou-exclusivo é facilmente implementada utilizando o operador $\wedge$, que realiza a operação bit a bit entre dois tipos primitivos. Estes fragmentos são então armazenados em diferentes repositórios.

O *broker* permite ainda utilizar o algoritmo de dispersão de informação (IDA). Utilizamos o algoritmo proposto por Malluhi e Johnston [87], que possui uma complexidade computacional menor que a versão original proposta por Rabin [101]. Para realizar as operações, utilizamos um Campo de Galois de 256 elementos, gerado a partir da potência 8 do número primo 2. Deste modo, podemos representar cada elemento por um byte. Os resultados das operações de soma neste campo são obtidos por uma operação de ou-exclusivo, enquanto que as de multiplicação são obtidas através de uma tabela de tamanho 256x256, populada quando a biblioteca é carregada. Detalhes dos algoritmos de codificação e decodificação são fornecidos na Seção 2.4.1.

3.5 Aplicações paralelas BSP

Realizamos algumas alterações na BSPLib do InteGrade [57], descrita na Seção 2.3, para permitir que aplicações BSP possam ser reiniciadas a partir de um *checkpoint* anterior. A primeira alteração foi no modo através do qual processos obtêm o seu `pid`, que identifica os processos de uma aplicação na BSPLib. Quando reiniciamos uma aplicação BSP, é necessário que os processos recebam o mesmo `pid` da execução anterior. Após a modificação, estes `pids`, que antes eram distribuídos aleatoriamente, passaram a ser atribuídos de maneira determinística. Além disso, implementamos duas extensões para a BSPLib, que permitem que (1) os processos de uma aplicação executem em máquinas heterogêneas e (2) o estado da aplicação seja salvo em um checkpoint.

3.5.1 Execução de aplicações BSP em máquinas de diferentes arquiteturas

Para permitir que a comunicação DRMA seja feita de maneira portátil, incluímos na assinatura do método `bsp_push_reg` um parâmetro que descreve o tipo de dado contido no endereço sendo registrado. Quando um processo remoto escreve em um endereço virtual registrado, o processo que recebe os dados compara a arquitetura onde cada processo está sendo executado e, se necessário, realiza a conversão entre os tipos de dados. Para tratar o caso de comunicação BSMP, modificamos a função `bsp_move`, incluindo como parâmetro extra o tipo de dado a ser lido do *buffer*. Novamente, a conversão é realizada apenas se a arquitetura da máquina que enviou a mensagem for diferente.

O pré-compilador obtém o tipo dos dados contidos no endereço passado como parâmetro às chamadas a `bsp_push_reg` e `bsp_move`. As chamadas a estes métodos são então modificadas, adicionando como parâmetro extra o tipo de dado sendo registrado ou transmitido. No exemplo da aplicação BSP que calcula médias da Seção 2.3, o pré-compilador modificaria a chamada a `bsp_push_reg`, na linha 12, que passariam a ser:

```
bsp_push_reg(&lista_de_medias, n_processos*sizeof(double), CKPT_DOUBLE);
```

3.5.2 Checkpointing de aplicações BSP

Na API da BSPlib, o controle da execução da aplicação é determinado pelas funções `bsp_begin`, que determina o início da seção paralela de uma aplicação BSP, `bsp_sync`, que determina a fase de sincronização e `bsp_end`, que identifica o fim da seção paralela.

Para permitir que a biblioteca de *checkpointing* salve o estado de aplicações BSP, adicionamos a esta biblioteca um módulo de coordenação para realizar o *checkpointing* da aplicação. Este módulo contém versões estendidas das funções `bsp_begin` e `bsp_sync`, que realizam a coordenação do processo de *checkpointing*. A versão estendida de `bsp_begin`, agora denominada `bsp_begin_ckp`, além das função já realizadas por `bsp_begin`, inicializa o protocolo coordenado de *checkpointing*. Do mesmo modo, a função `bsp_sync_ckp`, além das funções já realizadas por `bsp_sync`, é responsável pela coordenação do protocolo de *checkpointing*, garantindo que sejam sempre gerados estados globais consistentes. Este método é também responsável pelo gerenciamento de checkpoints obsoletos.

A substituição dos métodos da BSPlib pelos novos métodos é feito transparentemente pelo pré-compilador do mecanismo de *checkpointing*. Além disso, durante a reinicialização da aplicação, as funções que modificam o estado interno da biblioteca BSPlib, tais como `bsp_push_reg` e as versões estendidas das funções `bsp_begin`, `bsp_sync` e `bsp_end`, precisam ser executadas novamente. O pré-compilador instrumenta o código fonte de modo a garantir a execução destas funções durante a reinicialização.

3.6 Resumo

Neste capítulo, apresentamos a implementação do mecanismo de recuperação por retrocesso baseado em *checkpointing* no InteGrade, incluindo o uso de repositórios distribuídos em um mesmo aglomerado para armazenar os dados de *checkpointing*. No próximo capítulo descreveremos diversos experimentos com esta infra-estrutura, avaliando seu impacto no tempo de execução de aplicações.

Capítulo 4

Avaliação do Mecanismo de *Checkpointing*

Para avaliar a abordagem de *checkpointing* adotada, realizamos experimentos utilizando duas aplicações baseadas no modelo de programação BSP e que utilizam a biblioteca BSPlib do InteGrade:

- A aplicação de *similaridade de seqüências* [4, 60] calcula a similaridade entre duas seqüências de caracteres, baseada em um determinado critério. Esta aplicação é utilizada tanto no domínio da biologia, para realizar a comparação entre seqüências genéticas, como em aplicações de comparação de arquivos e de seqüências de caracteres. Para um par de seqüências de tamanhos m e n , a aplicação necessita de $O((m * n)/p)$ de memória em cada processo e $O(m * n/p)$ passos computacionais por processo. Além disso, são utilizadas $O(p)$ rodadas de comunicação por seqüência comparada, onde cada processo transfere $O(m/p)$ bytes por rodada de comunicação.
- A aplicação de *multiplicação de matrizes* [60] realiza a multiplicação de uma seqüência de m matrizes. Para realizar o cálculo, a aplicação divide a matriz em p partes, onde p é o número de processadores. Para uma matriz de tamanho $n \times n$, esta aplicação requer $O(n^2/p)$ de memória por processo e $O(m * \sqrt{p})$ rodadas de comunicação, onde cada processo envia $O(n^2/p)$ bytes por rodada de comunicação. Conseqüentemente, a aplicação de multiplicação requer a transferência de uma maior quantidade de dados entre processos e gera *checkpoints* de maior tamanho.

Ambas as aplicações foram desenvolvidas em C pelo então mestrando Ulisses Hayashida, membro do projeto InteGrade. As aplicações foram processadas pelo pré-compilador do mecanismo de *checkpointing*, descrito na Seção 3.4.1.

Para a realização dos experimentos, utilizamos um aglomerado contendo 10 máquinas AthlonXP 1700+ com 1GB de RAM e sistema operacional Linux 2.6, conectadas por uma rede Fast Ethernet chaveada de 100Mbps. Estas máquinas são utilizadas por estudantes no período diurno e permanecem ociosas durante o período noturno. Por este motivo, escolhemos o período noturno para a realização dos experimentos. Os principais resultados apresentados neste capítulo foram publicados em artigos [37, 42].

Iniciamos o capítulo com experimentos que medem o custo adicional imposto pelo mecanismo de *checkpointing* ao tempo de execução de uma aplicação. Em seguida, na Seção 4.2 avaliamos o mecanismo de recuperação por retrocesso do InteGrade, medindo o tempo necessário para uma aplicação finalizar sua execução na presença de falhas. Na Seção 4.3, analisamos a execução de aplicações BSP em nós de arquiteturas heterogêneas. Na Seção 4.4, medimos o tempo necessário para realizar a conversão entre dados de diferentes arquiteturas. Finalmente, na Seção 4.5, avaliamos as diferentes estratégias para armazenamento de *checkpoints* em um aglomerado.

4.1 Sobrecarga causada pelo mecanismo de *checkpointing*

Avaliamos a sobrecarga sobre o tempo de execução de uma aplicação paralela causada pelo processo de *checkpointing* utilizando diferentes valores de intervalos mínimos entre checkpoints. Comparamos também a influência do local de armazenamento dos *checkpoints* sobre esta sobrecarga. Neste experimento, armazenamos uma cópia simples do *checkpoint* gerado. Deixamos a avaliação de estratégias de armazenamento redundante para a Seção 4.5.

Utilizamos os seguintes locais para armazenamento de *checkpoints*:

- *sistema de arquivos local*: a biblioteca de *checkpointing* armazena os *checkpoints* em um diretório localizado no sistema de arquivos do disco local;
- *sistema de arquivos de rede NFS*: os *checkpoints* são armazenados em um diretório gerenciado por um servidor de arquivos NFS, compartilhado entre todas as máquinas do aglomerado. Neste cenário, os dados são transmitidos utilizando o protocolo interno do NFS;
- *repositório remoto de checkpoints*: uma máquina é selecionada para servir exclusivamente como repositório de *checkpoints*. Neste cenário, a biblioteca de *checkpointing* se conecta ao repositório remoto e transmite os checkpoints utilizando soquetes e o protocolo TCP/IP.

Realizamos também a execução da aplicação utilizando o código original da aplicação, isto é, sem a criação periódica de *checkpoints*.

Similaridade de seqüências

Para a aplicação de similaridade de seqüências, realizamos experimentos utilizando os métodos de armazenagem acima citados, cada um com 3 intervalos mínimos entre checkpoints: 10s, 30s, e 60s. Executamos a aplicação 5 vezes para cada cenário, utilizando como entrada 8 pares de seqüências de 100 mil caracteres. Foram utilizados 8 nós em cada execução. Os resultados são apresentados na Tabela 4.1.

Os *checkpoints* gerados por cada processo são de tamanho 125k bytes. t_{min} representa o intervalo mínimo ente *checkpoints*, n_{ckp} representa o número de *checkpoints* gerados, t_{nfs} a média do tempo

t_{min}	n_{ckp}	t_{nfs}	t_{local}	t_{orig}
10s	35	497,19s	492,96s	489,75s
30s	14	496,12s	492,36s	489,75s
60s	7	493,40s	491,45s	489,75s

Tabela 4.1: Sobrecarga do processo de *checkpointing* para a aplicação de similaridade de seqüências.

de execução para *checkpoints* armazenados no NFS, t_{local} para *checkpoints* armazenados no sistema de arquivos local e t_{orig} para o caso em que *checkpoints* não são gerados. Neste experimento não utilizamos o repositório remoto.

Em todos os casos, mesmo utilizando um intervalo de *checkpointing* de 10s, a sobrecarga causada ficou abaixo de 2%. Havíamos inicialmente obtido resultados que indicavam uma sobrecarga da ordem de 10%, que pudemos reduzir após otimizarmos a biblioteca de *checkpointing* de modo a criar um *thread* independente para armazenar os *checkpoints* gerados no sistema de arquivos. Na versão anterior da biblioteca, a aplicação ficava bloqueada enquanto os dados eram salvos. Observamos ainda que o armazenamento dos arquivos de *checkpoint* no sistema de arquivos local produziu metade da sobrecarga que o armazenamento com NFS.

Multiplicação de matrizes

Para a aplicação de multiplicação de matrizes, utilizamos os mesmos cenários da aplicação de similaridade de seqüências. Configuramos a aplicação para utilizar 9 nós, com matrizes contendo 450x450, 900x900 e 1800x1800 elementos do tipo `long double`. As execuções utilizando estas matrizes geram *checkpoints* globais de tamanho 2,3MB, 9,3MB e 37,1MB, respectivamente. Para cada tamanho de matriz, realizamos uma série de multiplicações: 300, 40 e 6, respectivamente. Para cada caso, executamos a aplicação 5 vezes. Devido a flutuações presentes na rede de estações de trabalho compartilhadas, consideramos apenas a média entre as duas execuções com menor tempo para cada caso. Supomos, neste caso, que as execuções realizadas em menor tempo foram as que sofreram menor influência de fatores externos, como atividades na rede e nas máquinas não relacionadas à execução da aplicação.

A Tabela 4.2 mostra os resultados obtidos para as matrizes de tamanho 450x450 e 1800x1800. ckp_{int} representa o intervalo mínimo entre checkpoints, n_{ckp} representa o número de *checkpoints* gerados e t_{orig} , t_{nfs} , t_{local} e t_{repos} representam o tempo total de execução da aplicação utilizando os diferentes locais de armazenamento.

Os resultados mostram que salvar os *checkpoints* no sistema de arquivos local ou em um repositório remoto é ligeiramente mais eficiente que utilizar o NFS. O desempenho inferior do NFS provavelmente se deve às políticas de *caching* e ao controle extra necessário para manter um sistema de arquivos. Quando utilizamos um repositório remoto, a sobrecarga de *checkpointing* ficou consistentemente abaixo de 10%, mesmo para matrizes de 1800x1800 e um intervalo de *checkpointing* de 10s. Em comparação com a

tamanho da matriz = 450x450					
ckp_{int}	n_{ckp}	t_{orig}	t_{nfs}	t_{local}	t_{repos}
10s	12	122,8s	136,0s	134,3s	131,5s
30s	4	122,8s	124,3s	128,5s	126,7s
60s	2	122,8s	125,8s	123,4s	125,5s

tamanho da matriz = 1800x1800					
ckp_{int}	n_{ckp}	t_{orig}	t_{nfs}	t_{local}	t_{repos}
10s	10	165,2s	188,8s	176,8s	176,3s
30s	4	165,2s	195,1s	172,7s	171,0s
60s	2	165,2s	170,2s	168,1s	169,0s

Tabela 4.2: Tempos de execução para a aplicação de multiplicação de matrizes.

aplicação de similaridade de seqüência, uma vez que é necessário salvar grandes quantidades de dados, o processo de *checkpointing* causa uma sobrecarga maior. Apesar disso, os resultados obtidos indicam que é possível utilizar nosso mecanismo de *checkpointing* para aplicações que utilizam uma quantidade de memória da ordem de dezenas de Megabytes.

4.2 Execução de uma aplicação BSP na presença de falhas

O objetivo deste experimento é simular um ambiente dinâmico onde recursos podem ficar indisponíveis a qualquer momento. Para cada nó da grade, geramos uma seqüência de tempos representando os momentos em que aquele nó se torna indisponível. Durante sua inicialização, cada LRM lê esta seqüência de um arquivo e a cada vez que um tempo presente nesta seqüência é atingido, o LRM envia um sinal do tipo KILL a todos os processos da grade sendo executados em sua máquina.

Para gerar os tempos para a próxima falha, utilizamos a distribuição exponencial [64]. De acordo com este modelo, a distribuição de tempos entre falhas é dado por:

$$D(t) = 1 - e^{-\lambda t}$$

onde $1/\lambda$ representa o tempo médio entre falhas.

Neste experimento, simulamos dois cenários com diferentes tempos médios entre falhas nas máquinas: 600s e 1800s. Para ambos os cenários, utilizamos um intervalo mínimo entre *checkpoints* de 10, 30 e 60 segundos, armazenando os *checkpoints* gerados num sistema de arquivos NFS compartilhado entre os nós. Em todos os casos, executamos a aplicação utilizando 9 nós.

Executamos a aplicação de similaridade de seqüências 5 vezes para cada caso, utilizando uma seqüência de tempos entre falhas distinta para cada execução, mas mantendo as mesmas seqüências para intervalos de *checkpointing* distintos. Deste modo, reduzimos o grau de variabilidade entre os experimentos. Os resultados são apresentados na Tabela 4.3.

$ckp_{interval}$	$1/\lambda$	t_{total}
10s	600s	517,4s
30s	600s	571,2s
60s	600s	699,0s
10s	1800s	490,3s
30s	1800s	519,3s
60s	1800s	534,5s

Tabela 4.3: Tempo de execução da aplicação de similaridade de seqüências na presença de falhas. LRMs foram configurados para simular uma falha com um tempo médio entre falhas ($1/\lambda$) de 600s e 1800s. Intervalos mínimos entre *checkpoints* ($ckp_{interval}$) de 10s, 30s, e 60s foram utilizados. t_{total} representa o tempo total de execução da aplicação.

Os resultados indicam que um intervalo menor entre *checkpoints* reduz o tempo de execução total na presença de falhas. Isto ocorre porque, a cada reinicialização, uma menor quantidade de computação é perdida. Mas, neste experimento, utilizamos a aplicação de similaridade de seqüências, onde a sobrecarga gerada pelo processo de *checkpointing* é pequena.

Para aplicações onde a sobrecarga do processo de *checkpointing* é maior, é possível que a menor quantidade de computação perdida durante a reinicialização não compense a sobrecarga extra causada pelo menor intervalo entre checkpoints. Um trabalho futuro é investigar este comportamento de modo a determinar se existem pontos de equilíbrio entre a sobrecarga de *checkpointing* e o tempo esperado para a próxima falha.

4.3 Execução de uma aplicação BSP em nós heterogêneos

Neste experimento, executamos a aplicação de multiplicação de matrizes em nós com diferentes arquiteturas. O objetivo foi determinar o impacto da heterogeneidade no tempo de execução da aplicação. Nesta aplicação, a matriz é dividida em submatrizes, que são atribuídas para cada um dos processos. Em cada super-passo do modelo BSP, os processos realizam operações sobre suas respectivas submatrizes e enviam para seus vizinhos os valores atualizados de parte dos elementos de sua submatriz.

Medimos o tempo necessário para transferir uma seqüência de bytes recebidos da rede para o espaço de memória da aplicação BSP, incluindo, se necessário, as conversões nas representações dos dados. Utilizamos uma configuração composta por 3 Athlons + 1 PowerPC G4, de modo que a aplicação foi configurada para ser executada com 4 processos. Os tempos foram medidos em um dos processos sendo executado na arquitetura x86. Este processo recebe elementos de matrizes de um processo executando sobre a arquitetura x86 e outro executando em uma arquitetura PowerPC.

Utilizamos dois tipos de dados para representar os elementos da matriz, `float` e `long double`. A conversão do tipo `float` é relativamente simples, bastando inverter a ordem dos bytes. No caso do tipo `long double`, no entanto, a conversão é mais complexa, uma vez que este tipo é representado de

maneira distinta nas duas arquiteturas (12 bytes no Linux com GCC3.3 e 8 bytes no MacOS X com GCC 3.1). Os resultados são apresentados na Tabela 4.4.

Tamanho da matriz	Tipo de elemento	t_{exec}	t_{comm}	t_{x86}	t_{ppc}
500x500	float	17,4s	0,132s	0,015s	0,112s
1000x1000	float	79,3s	0,251s	0,027s	0,220s
2000x2000	float	201,7s	0,202s	0,023s	0,177s
500x500	long double	28,8s	0,263s	0,042s	0,217s
1000x1000	long double	156,8s	0,510s	0,078s	0,430s
2000x2000	long double	373,5s	0,416s	0,066s	0,348s

Tabela 4.4: Tempo necessário para transferir uma sequência de bytes recebidos da rede para o espaço de memória da aplicação BSP. t_{exec} representa o tempo de execução total do processo, t_{comm} o tempo total utilizado para copiar os dados recebidos para o espaço de memória do processo, t_{x86} o tempo para copiar elementos da matriz enviados pelo processo executando na arquitetura x86 e t_{ppc} o tempo para copiar elementos da matriz enviado pelo processo executando no PowerPC.

Converter dados provenientes de uma sequência de bytes enquanto estes dados são copiados é muito mais caro que uma simples cópia desta sequência de bytes. Na Tabela 4.4 é possível observar que a cópia com conversão t_{ppc} demora até 8 vezes mais do que a cópia simples t_{x86} . O tempo gasto realizando as conversões é, no entanto, praticamente imperceptível quando comparado ao tempo total de execução da aplicação, o que indica que a solução que implementamos para a execução de aplicações BSP em nós heterogêneos é viável.

4.4 Recuperação do estado contido em um checkpoint

Comparamos o tempo necessário para reiniciar uma aplicação utilizando um *checkpoint* gerado em máquinas com diferentes arquiteturas. As arquiteturas selecionadas para este experimento foram x86 (Athlon), x86_64 (Athlon64) e PowerPC G4.

Utilizamos uma aplicação que gera uma lista ligada de estruturas contendo 20 mil nós, com cada nó contendo um número de tipo `long double` e ponteiros para outros dois nós. O primeiro ponteiro referencia o próximo nó da lista ligada, enquanto o outro ponteiro referencia um nó aleatório. Deste modo, os nós formam um grafo contendo diversos ciclos. Decidimos utilizar o tipo `long double` devido ao fato deste ser o tipo de dado que mais difere entre as arquiteturas selecionadas.

Os experimentos de reinicialização foram realizados em um Athlon 2.0GHz, utilizando checkpoints gerados por processos executando em máquinas Athlon, Athlon64 e PowerPC G4. Os resultados são apresentados na Tabela 4.5. Os valores representam o tempo gasto para recuperar o estado de um processo a partir dos dados contidos em um *checkpoint* já carregado em um *buffer* temporário. Este tempo inclui as conversões que precisam ser realizadas para obter os valores de variáveis locais e blocos de memória dinâmica a partir do checkpoint.

Arquitetura de origem	Tempo de Conversão	Sobrecarga
x86	0,179s	-
x86_64	0,186s	3.9%
PowerPC	0,192s	7.2%

Tabela 4.5: Tempo para a conversão e sobrecarga relativa à recuperação dos dados contidos em um *checkpoint* para uma aplicação que contém uma lista ligada de 20 mil estruturas.

Para ambas as arquiteturas, x86_64 e PowerPC, a sobrecarga causada pelas transformações de dados foi inferior a 10%. Uma vez que apenas uma pequena fração do tempo de execução de uma aplicação é utilizada durante a reinicialização, esta sobrecarga é praticamente imperceptível. Se lembrarmos que no processo de gerar *checkpoints* não é realizada nenhum tipo de conversão de dados, vemos que o custo de realizar conversões de dados pode ser considerado desprezível na prática.

4.5 Estratégias de armazenamento

Nestes experimentos, avaliamos as diferentes estratégias de armazenamento de *checkpoints* em um aglomerado. Utilizamos a aplicação de multiplicação de matrizes, com matrizes de diferentes tamanhos, compostas por elementos do tipo `long double`. O objetivo dos experimentos foi medir o tempo de codificação e decodificação de arquivos e a sobrecarga causada pelo armazenamento de *checkpoints* utilizando as diferentes estratégias de armazenamento.

4.5.1 Codificação e decodificação de dados

Neste experimento medimos o tempo necessário para codificar e decodificar o conteúdo de um arquivo utilizando os algoritmos IDA e de paridade local. Para a comparação, utilizamos diversos tamanhos de dados e número de fragmentos. Os resultados obtidos são mostrados na Figura 4.1. Nos gráficos, $IDA(m, k)$, representa o algoritmo IDA utilizando $m + k$ fragmentos, dos quais m são suficientes para reconstruir o *checkpoint*, como descrito na Seção 2.4.1.

Como esperado, o cálculo de paridade local é mais rápido que IDA em todos os cenários. O resultado mais interessante, é que a codificação utilizando IDA não é tão cara quanto esperado. A codificação de 100Mbytes de dados requer apenas poucos segundos, com o tempo gasto com a codificação aumentando linearmente com número de fragmentos redundantes (k) e quantidade de dados. Os resultados deste experimentos foram bastante satisfatórios.

Já a decodificação é mais rápida que a codificação. Isto porque na codificação é necessário gerar $m + k$ fragmentos, enquanto na decodificação é preciso reconstruir o arquivo original a partir de apenas m fragmentos.

Figura 4.1: Tempo para codificar e decodificar um arquivo.

4.5.2 Sobrecarga sobre o tempo de execução

Avaliamos também a sobrecarga causada pelos processos de *checkpointing*, codificação e armazenamento distribuído sobre o tempo de execução de aplicações paralelas. O objetivo deste experimento é comparar a sobrecarga das diversas estratégias de armazenamento descritas na Seção 2.4.1. Utilizamos 8 nós para executar a aplicação, composta por 16 processos, e dois tamanhos para a matriz: 3200x3200 e 4800x4800. Decidimos armazenar os *checkpoints* gerados nas máquinas que executavam os processos da aplicação para avaliar o impacto do armazenamento de *checkpoints* sobre o tempo de execução da aplicação. Analisamos os seguinte cenários:

1. *Aplicação original*: *checkpoints* são gerados mas não são armazenados;
2. *Replicação*: uma cópia do *checkpoint* é armazenada localmente e outra em uma das máquinas executando os demais processos da aplicação;
3. *Paridade*: o *checkpoint* é dividido em 7 fragmentos e é gerado um fragmento extra contendo

informações de paridade. Os fragmentos são então armazenados nas máquinas executando os processos da aplicação;

4. $IDA(m=7,k=1)$: o *checkpoint* é codificado em 8 fragmentos, dos quais 7 são suficientes para recuperá-lo. Os fragmentos são então armazenados nas máquinas executando os processos da aplicação;
5. $IDA(m=6,k=2)$: o *checkpoint* é codificado em 8 fragmentos, dos quais 6 são suficientes para recuperá-lo. Os fragmentos são então armazenados nas máquinas executando os processos da aplicação.

A Tabela 4.6 mostra os parâmetros de execução utilizados no experimento. Representamos os tamanhos dos *checkpoints* local e global por $tamanho_{local}$ e $tamanho_{global}$ respectivamente. $t_{original}$ representa o tempo necessário para executar o código original da aplicação, sem a instrumentação. Finalmente, definimos como 60s o intervalo mínimo entre checkpoints.

Tabela 4.6: Parâmetros de execução.

tamanho da matriz	$t_{original}$	$tamanho_{local}$	$tamanho_{global}$
3200x3200	1115,1s	22,0MB	351,6MB
4800x4800	1101,6s	49,4MB	791,0MB

Na Figura 4.2, apresentamos a sobrecarga do uso de *checkpointing* utilizando diferentes estratégias de armazenamento e diferentes tamanhos de matrizes. Para cada caso, realizamos 16 execuções da aplicação e calculamos a média e o desvio padrão do tempo de execução. O eixo x contém os 5 cenários para a estratégia de armazenamento apresentados anteriormente. O eixo y contém o tempo de execução médio para cada cenário e as barras representam o desvio padrão. Os valores $nCkp = \dots$ representam o número médio de *checkpoints* gerados durante a execução da aplicação em cada cenário.

Os resultados mostram que a utilização de IDA (cenários 4 e 5) causa a maior sobrecarga. Isto era esperado, dado que é necessário realizar a codificação dos dados. No caso de matrizes de tamanho 4800x4800, onde são gerados *checkpoints* globais de 791MB com um intervalo mínimo de 60s, a sobrecarga é de aproximadamente 20% para a estratégia $IDA(m=6,k=2)$. Podemos facilmente reduzir esta sobrecarga para valores próximos a 2% aumentando o intervalo entre *checkpoints* para 10 minutos.

Utilizando as estratégias de paridade ou replicação é possível obter sobrecargas menores, mas com um nível de tolerância a falhas menor. Por outro lado, utilizando IDA podemos manipular os parâmetros m e k , obtendo o balanceamento desejado entre rapidez de execução, utilização de recursos e nível de tolerância a falhas.

Figura 4.2: Sobrecarga do armazenamento de *checkpoints* sobre o tempo de execução da aplicação de multiplicação de matrizes.

4.6 Resumo

Neste capítulo analisamos experimentalmente a abordagem que adotamos para gerar e armazenar checkpoints. Como vimos, para aplicações que geram *checkpoints* globais com tamanhos de 791MB a cada 60s, a sobrecarga fica entre 10% e 20%, mas esta pode ser facilmente reduzida aumentando o intervalo entre *checkpoints* para 5 ou 10 minutos. Estes resultados indicam que a utilização de um mecanismo de *checkpointing* portátil e o armazenamento dos *checkpoints* nos nós de processamento da grade são viáveis numa grade oportunista.

Mas as aplicações não dependem apenas de arquivos de checkpoints, mas também de arquivos de entrada e saída. Neste caso, seria interessante termos um mecanismo mais genérico para armazenar os diferentes tipos de arquivos. Além disso, o armazenamento de *checkpoints* ficou restrito a um único aglomerado. Numa grade, é importante que o armazenamento e acesso a *checkpoints* e dados de entrada e saída de aplicações possam ser realizados de qualquer ponto da grade, fornecendo uma flexibilidade muito maior ao sistema de grade para escalonar execuções em diferentes aglomerados. Finalmente, armazenar todos os dados de uma aplicação em um único aglomerado implica que uma falha neste aglomerado causará a indisponibilidade destes dados.

Nos próximos capítulos analisaremos possíveis soluções para estes problemas e apresentaremos o

middleware OppStore, que realiza o armazenamento distribuído dos dados de aplicações nas máquinas compartilhadas de uma grade oportunista, distribuídas em aglomerados distintos.

Capítulo 5

Armazenamento Distribuído de Dados

Até este ponto da tese, tratamos do mecanismo de *checkpointing*, que permite a execução robusta de aplicações paralelas em máquinas não-dedicadas. Tratamos também do armazenamento dos *checkpoints* gerados nos aglomerados onde as aplicações são executadas. Mas, além de *checkpoints*, aplicações de diversas áreas, como bioinformática, mineração de dados e processamento de imagens, normalmente precisam manipular diversos outros tipos de dados, como arquivos de entrada e saída. Além disso, para armazenar os dados de modo mais confiável e facilitar o acesso a estes dados de qualquer ponto da grade, o ideal seria distribuir estes dados também em outros aglomerados. Para solucionar estes problemas, optamos por desenvolver um sistema de armazenamento distribuído de dados para grades oportunistas.

Um requisito importante para o nosso sistema de armazenamento é que ele não deve exigir a aquisição de hardware especializado. Estamos considerando que o sistema irá operar sobre uma grade oportunista, cujo foco é o uso de recursos ociosos de máquinas já existentes. A idéia é permitir que instituições com restrições orçamentárias possam usufruir dos recursos computacionais que já possuem.

O sistema de armazenamento é destinado a aplicações científicas que possam ser executadas em grades oportunistas. As máquinas deste tipo de grade possuem limitações de desempenho e não são apropriadas para executar aplicações que requerem Petabytes de dados, que precisariam ser acessíveis através redes de alta velocidade que transportam centenas de Gigabytes por segundo. Este tipo de aplicação pode requerer infra-estruturas de muitos milhões de dólares. Neste trabalho, o objetivo é permitir o armazenamento de dados sem custos extras significativos para as instituições interessadas.

A partir destes requisitos e suposições, propomos a utilização do espaço livre em disco dos nós provedores de recurso da grade para realizar o armazenamento de dados. Estas são normalmente máquinas compartilhadas, que devem ser utilizadas para transferir dados apenas durante seus períodos ociosos. Nos demais períodos, estas máquinas devem ser consideradas indisponíveis. O sistema de armazenamento deve ser capaz de lidar com este ambiente altamente dinâmico, fornecendo um armazenamento eficiente e confiável para dados de aplicações.

Um ambiente similar, onde o espaço em disco disponível de computadores pessoais é utilizado para

armazenar e distribuir arquivos, é explorado pelos sistemas *peer-to-peer* de distribuição de conteúdo [8]. Mais recentemente, sistemas de arquivos distribuídos altamente escaláveis e flexíveis, baseados em *peer-to-peer*, têm sido desenvolvidos, como CFS [35] e PAST [108]. Estes sistemas obtiveram bons resultados, indicando que o uso de redes sobrepostas *peer-to-peer* para organizar as máquinas pode ser uma escolha apropriada para armazenar dados em grades oportunistas. Por outro lado, outros estudos indicam que a escalabilidade destes sistemas pode ser limitada [18].

Iniciamos o capítulo apresentando os conceitos de redes *peer-to-peer* e de tabelas de espalhamento distribuídas, incluindo suas principais características e limitações. Em seguida, na Seção 5.2, discutimos os principais sistemas de armazenamento distribuído baseados em redes *peer-to-peer*. Discutimos então, na Seção 5.3, as principais abordagens utilizadas atualmente para o armazenamento de dados em grades computacionais. Finalizamos o capítulo com um resumo e breve discussão dos conceitos apresentados.

5.1 Sistemas *peer-to-peer*

Sistemas *peer-to-peer* [8, 84, 120, 122, 125, 127] são formados por um conjunto de máquinas (*peers*), que se conectam constituindo uma rede sobreposta, onde cada máquina assume tarefas tanto de cliente como de servidor. Consideramos um sistema *peer-to-peer* como puro quando todas as máquinas conectadas ao sistema realizam as mesmas funções. Uma importante vantagem dos sistemas *peer-to-peer* é que suas funcionalidades são distribuídas pelos nós da rede de maneira descentralizada, o que contribui para permitir a criação de sistemas escaláveis, auto-organizáveis e tolerantes a falhas. Sistemas *peer-to-peer* são amplamente utilizados hoje na Internet na criação de redes de distribuição de conteúdo [8, 84].

As redes sobrepostas que constituem os sistemas *peer-to-peer* são classificadas como *estruturadas* ou *não-estruturadas*. Nas redes estruturadas, os nós são conectados de acordo com regras rígidas de organização e a determinação do local de armazenamento de dados é feita de modo determinístico. Já nas redes não-estruturadas, os nós se conectam de acordo com regras muito mais flexíveis, além de possuir mais autonomia para escolher quais dados armazenar. Dadas estas características, podemos imaginar que as redes estruturadas permitem uma localização mais eficiente dos dados armazenados ao custo de uma menor flexibilidade e tolerância a falhas.

Tabelas de espalhamento distribuídas (*Distributed Hash Tables* ou DHTs) [104, 109, 118, 145] armazenam pares (chave, dados), onde a chave identifica algum conjunto de dados armazenado. Estes dados são distribuídos em diversos nós, onde cada nó é responsável por uma fatia do espaço de identificadores (chaves). Para aumentar a escalabilidade, a tabela contendo os pares (chave, dados) também é distribuída através dos nós. Normalmente, cada nó mantém os pares relativos ao seu espaço de identificadores. Redes *peer-to-peer* estruturadas normalmente implementam a funcionalidade de uma DHT. Um cliente que deseje armazenar ou recuperar um conjunto de dados no sistema normalmente envia uma requisição ao sistema com o identificador destes dados. O sistema realiza então o roteamento da requisição até o nó responsável por este identificador. Este roteamento é realizado com cada nó repassando a requisição para

outros nós até o nó de destino ser encontrado.

Já as redes não-estruturadas normalmente implementam uma busca baseada em pesquisas (*queries*). Mensagens contendo os termos pesquisados são distribuídas entre os nós até que um ou mais nós contendo o objeto da pesquisa sejam encontrados.

Nesta seção, descreveremos algumas implementações de redes *peer-to-peer* não-estruturadas e suas principais características. Em seguida, discutiremos as tabelas de espalhamento distribuídas (DHTs), que são a base para as redes *peer-to-peer* estruturadas. Finalizamos a seção abordando o problema do balanceamento de carga em DHTs e algumas das soluções propostas.

5.1.1 Redes *peer-to-peer* não-estruturadas

Nas redes *peer-to-peer* não-estruturadas, os nós se organizam de maneira aleatória, sem uma estrutura global pré-definida. Para realizar a busca, os nós se utilizam de técnicas como a inundação (*flooding*) [65], possivelmente com um marcador limitando a distância da busca, ou caminhadas aleatórias. Estas técnicas permitem que cada nó verifique se ele possui algum conteúdo digital que possa ser retornado pela busca, que pode incluir expressões complexas. Por outro lado, a busca é ineficiente, uma vez que os nós da busca são escolhidos de modo aleatório. Para resolver este problema, alguns sistemas se utilizam de nós que realizam a indexação dos dados armazenados no sistema.

Foram desenvolvidos diversos sistemas *peer-to-peer* para a distribuição de conteúdo digital baseados em redes não-estruturadas. Muitos estão atualmente em operação sobre a Internet, sendo compostos por milhares ou até milhões de nós distribuídos pelo mundo. Entre os sistemas baseados em redes não estruturadas estão o Napster [127], Gnutella [122], FastTrack (utilizado por aplicativos como o Kazaa [126]) e Freenet [120].

O Napster foi o primeiro sistema *peer-to-peer* destinado ao compartilhamento de dados. Apesar dos dados serem transmitidos diretamente entre os nós clientes, o Napster se utiliza de um diretório centralizado para indexar conteúdo dos clientes. Esta centralização gera um ponto único de falha, além de servir como gargalo à sua escalabilidade.

Diferentemente do Napster, o Gnutella foi concebido como um sistema *peer-to-peer* totalmente descentralizado, onde cada nó do sistema realiza exatamente as mesmas tarefas. A topologia da rede é aleatória e o sistema não possui controles globais para indexação e busca de conteúdos armazenados. Esta busca é feita por um mecanismo de inundação, com um contador que limita o raio da busca. Apesar de ser altamente tolerante a falhas, o mecanismo de busca baseado em inundação pode facilmente sobrecarregar os nós, limitando a escalabilidade do sistema. Além disso, arquivos raros dificilmente são encontrados pelas buscas.

O FastTrack tenta amenizar o problema das buscas em Gnutella utilizando o conceito de super-nós. Estes são nós com alta capacidade de processamento e conectados por redes mais rápidas. Estes super-nós indexam o conteúdo de parte dos demais nós da rede, de modo que as buscas possam ser feitas nesta

rede menor formada por super-nós. Em compensação, a manutenção dos índices nos super-nós gera tráfego extra e as buscas ainda são feitas de modo aleatório.

O Freenet oferece a funcionalidade de uma tabela de espalhamento distribuída utilizando uma rede *peer-to-peer* não-estruturada como substrato. Suas principais características são a independência do local de armazenamento de um conteúdo digital com relação a sua chave e a possibilidade de armazenamento anônimo de informações. Durante as buscas, cada nó possui informações apenas sobre seus vizinhos imediatos e decide sobre o roteamento da mensagem utilizando esta informação local. Por não ter uma organização global rígida entre os nós, o tratamento de entrada, saída e falhas de nós é facilitado.

O principal motivo para a escolha de redes sobrepostas não-estruturadas é facilitar a busca por conteúdo digital a partir de palavras-chaves ou outros tipos de meta-dados. Outra vantagem destas redes é que a ausência de uma estrutura global rígida facilita a manutenção do sistema num cenário onde nós constantemente entram e saem da rede. Finalmente, a busca por conteúdo digital popular em redes não-estruturadas pode ser bastante eficiente, já que estes dados provavelmente estarão replicados em muitos nós.

A principal desvantagem destas redes é que conteúdos digitais pouco utilizados são difíceis de localizar e podem ser facilmente perdidos. Além disso, a maioria dos algoritmos de busca consomem muitos recursos de rede e processamento, podendo sobrecarregar o sistema. Existem diversos estudos em andamento buscando minimizar ou eliminar estas limitações dos sistemas baseados em redes não-estruturadas [65].

5.1.2 Redes *peer-to-peer* estruturadas

Em redes *peer-to-peer* estruturadas os nós são organizados numa estrutura global bem definida, onde os locais de armazenamento de dados são deterministicamente selecionados, de modo a facilitar a posterior busca por estes dados. Estas redes utilizam o conceito de tabelas de espalhamento distribuídas (DHTs), descritas anteriormente, para realizar o mapeamento entre chaves e locais de armazenamento dos dados associados àquela chave.

Existem diversas implementações de redes *peer-to-peer* estruturadas, como o Pastry [109], Tapestry [145], Bamboo [105], Chord [118], CAN [104], Kademlia [89] e Viceroy [86]. Nesta seção descrevemos brevemente três destes protocolos: Pastry, Chord e CAN. Na Seção 6.2 apresentaremos em mais detalhes o protocolo Pastry, que foi utilizado em nosso trabalho.

Pastry fornece uma infra-estrutura que implementa a funcionalidade de uma tabela de espalhamento distribuída. Cada nó em Pastry recebe um identificador de 128 bits e o sistema realiza o roteamento de uma mensagem para o nó com o identificador numericamente mais próximo ao identificador da mensagem. Pastry utiliza o conceito de roteamento por prefixo onde, a cada passo, a mensagem é encaminhada a um nó cujo identificador tenha um maior prefixo em comum com o identificador da mensagem. Para permitir o roteamento, Pastry mantém diversas estruturas de dados, incluindo a *tabela de roteamento*,

que permite o roteamento por prefixos das mensagens, o *conjunto de folhas*, que contém os vizinhos lógicos do nó, e o *conjunto de vizinhos*, que contém um conjunto de nós geograficamente próximos. Para um conjunto de n nós, o Pastry permite encontrar o nó responsável pelo identificador de uma mensagem em $\log_B n$ passos, onde B é um parâmetro que permite configurar o tamanho das tabelas dos nós.

Chord organiza os nós da rede em um anel lógico de modo que identificadores são mapeados para o primeiro nó sucessor daquele identificador no anel. Cada nó mantém uma lista com alguns nós sucessores. A busca é realizada percorrendo o anel até encontrar um nó cuja lista de sucessores contém o nó imediatamente sucessor ao identificador da mensagem. Para tornar esta busca eficiente, cada nó mantém uma tabela de apontadores (*finger table*) para nós espalhados pelo anel. Esta tabela contém no máximo m elementos com valores $2^{(i-1)}$, onde $1 \leq i \leq m$ e m é o número de bits do identificador, e permite que, a cada passo do roteamento, a distância da mensagem ao nó de destino seja dividida por dois. Conseqüentemente, Chord permite que uma mensagem chegue ao nó responsável por um identificador em $O(\log n)$ passos.

CAN (*Content Addressable Network*) organiza os identificadores em um espaço de d dimensões situado sobre um d-toróide. Cada nó em CAN é responsável por uma área deste espaço d-dimensional e chaves são mapeadas neste espaço através de uma função de espalhamento uniforme. Para permitir o roteamento de mensagens, cada nó mantém o endereço dos nós adjacentes à área de sua responsabilidade. Utilizando CAN, uma mensagem é roteada ao nó de destino em $O(d * n^{1/d})$ passos, onde n é o número de nós na rede e d a dimensão do espaço de coordenadas, de modo que o crescimento no número de passos é polinomial. Em compensação, o tamanho das tabelas alocadas para cada nó é independente do número total de nós no sistema.

Redes *peer-to-peer* estruturadas requerem que sejam definidos protocolos para entrada e saída de nós, uma vez que as tabelas de roteamento dos nós precisam ser atualizadas. Além disso, quando um nó entra ou sai do sistema, a faixa de identificadores alocada para os vizinhos daquele nó mudam, de modo que pode ser necessário transferir informações armazenadas entre os nós. Na Seção 6.2 discutiremos os protocolos de ingresso e saída de nós no Pastry, que foi o sistema no qual baseamos o nosso trabalho.

A grande vantagem das redes *peer-to-peer* estruturadas é que é possível encontrar, em poucos passos, a localização de qualquer conjunto de dados representado por uma chave. Esta garantia é essencial para algumas classes de aplicações, como sistemas de arquivos distribuídos. A desvantagem é que, como é necessário manter a estrutura da rede, esta é menos tolerante a falhas do que as redes não-estruturadas, além de exigir a manutenção de diversas tabelas nos nós. Além disso, a busca de dados é feita exclusivamente pela sua chave, não sendo possível o uso de palavras ou outros tipos de pesquisa mais sofisticadas.

5.1.3 Balanceamento de carga em DHTs

Um problema importante relativo às tabelas de espalhamento distribuídas (DHTs) é que, quando os identificadores dos nós são distribuídos de maneira aleatória, nós diferentes ficam responsáveis por faixas de identificadores de tamanhos diferentes. Dado um número m de identificadores e n de nós no sistema, é possível mostrar que com alta probabilidade existirão nós responsáveis por uma faixa de identificadores da ordem de $\log n$ vezes o tamanho médio da faixa alocada a cada nó [67].

A situação é agravada no caso de nós heterogêneos, já que o tamanho da faixa alocada a cada nó não tem nenhuma relação com a sua capacidade. Aqui estamos supondo que é possível definir uma métrica a partir da qual podemos estimar a capacidade de um nó para atender a requisições provenientes do sistema *peer-to-peer*. Por exemplo, em um sistema de armazenamento distribuído, a capacidade pode ser a largura de banda da rede que liga o nó ao restante do sistema ou quantidade de espaço disponível para armazenamento em disco. O resultado é que alguns nós da rede acabam sobrecarregados enquanto outros permanecem ociosos pela maior parte do tempo.

Normalmente, supomos que os identificadores dos dados são gerados de forma aleatória e uniforme, utilizando, por exemplo, uma função de espalhamento segura como SHA-1. Sob esta suposição, o balanceamento de carga seria equivalente a balancear o espaço de endereços. Mas ainda assim é possível que exista uma assimetria nas requisições a determinados identificadores. Por exemplo, um arquivo de um sistema de arquivos distribuídos pode ser particularmente popular, gerando uma sobrecarga extra em um servidor. Além disso, as taxas de acesso a cada arquivo tendem a variar dinamicamente, de modo que, durante a evolução do sistema, diferentes servidores ficarão sobrecarregados. Finalmente, a própria capacidade de um nó pode ser alterada com o passar do tempo. Esta volatilidade do sistema nos leva a outro conceito importante, que é o de *balanceamento de carga dinâmico*, onde a faixa de identificadores alocada para cada nó é alterada dinamicamente durante a execução do sistema. Deste modo, o sistema compensa as variações na carga ou capacidade de um nó que ocorrem durante o tempo de vida do sistema.

Técnicas para o balanceamento de carga

A abordagem mais utilizada para balancear a faixa de identificadores atribuída a cada nó é utilizar o conceito de *servidores virtuais* [67]. Neste caso, cada nó do sistema fica responsável pelas faixas de identificadores alocadas a um determinado número de servidores virtuais. Uma abordagem bastante simples é instanciar $O(\log n)$ servidores virtuais em cada nó, onde n é o número de nós na rede [35, 67, 118]. Deste modo, os nós passariam a ser responsáveis por faixas de endereços de aproximadamente o mesmo tamanho. Esta abordagem é bastante simples, mas tem a desvantagem de necessitar que cada nó armazene informações sobre $\log n$ nós ao invés de apenas um, além do tráfego gerado na rede para manter estas tabelas adicionais. No extremo oposto, existe a possibilidade de permitir que cada nó escolha e instancie exatamente um de $\log n$ servidores virtuais alocados para o nó [68]. Neste caso não

existe a sobrecarga causada pelo grande número de servidores virtuais.

Mas, ambas as abordagens acima citadas não tratam da heterogeneidade dos nós e não permitem a adaptação a mudanças dinâmicas na carga dos nós. Para tratar o problema da heterogeneidade, podemos permitir que cada nó escolha quais servidores virtuais instanciar de acordo com sua capacidade. Mas, neste caso, cada nó precisa comparar a sua capacidade com relação às de seus vizinhos e precisa de uma estimativa do número de nós presentes no sistema. Se servidores virtuais não fossem utilizados, o número de nós no sistema poderia ser estimado a partir do tamanho da faixa de identificadores coberta pelos nós vizinhos a um determinado nó. Esta estimativa é possível se supormos que os identificadores são fornecidos de forma aleatória. Mas se cada nó puder instanciar diferentes números de servidores virtuais baseados em sua capacidade, esta abordagem de estimativa local passa a ser inviável. Neste caso, com a utilização de servidores virtuais, pode ser difícil estimar o número de nós no sistema.

Uma maneira de solucionar este problema é utilizar o balanceamento de carga dinâmico, de modo que o sistema dinamicamente se ajuste de acordo com sua evolução. Para tal, podemos permitir que nós com alta utilização no sistema transfiram servidores virtuais para nós com baixa utilização. Podemos dizer que um nó está com alta utilização se o uso dos recursos do nó estiver acima de um determinado limiar e com baixa utilização se o uso dos recursos estiver abaixo deste mesmo limiar. Diversas estratégias podem ser utilizadas para selecionar quais nós devem trocar servidores virtuais [54, 103]. Uma maneira bastante simples é selecionar pares de nós aleatoriamente e realizar a troca de servidores virtuais se um nó estiver com alta e o outro com baixa utilização. Outra possibilidade é utilizar diretórios distribuídos que armazenam informações sobre a utilização dos nós do sistema e aos quais cada nó periodicamente envia seu estado. Nós com alta utilização podem procurar no diretório por nós com baixa utilização para o qual possam transferir servidores virtuais.

Uma abordagem diferente é permitir que os nós transfiram parte da sua faixa de identificadores para seus vizinhos [68, 144]. Esta distribuição de faixas de identificadores entre vizinhos pode levar em conta a heterogeneidade dos nós, de modo a atribuir a cada nó uma faixa de identificadores proporcional à sua capacidade. Deste modo, é possível fazer o balanceamento dinâmico de carga em redes heterogêneas utilizando apenas informação local sobre os vizinhos dos nós.

Um fator importante a se considerar é que mudanças nas faixas de identificadores alocadas a cada nó normalmente requerem que dados sejam transferidos entre estes nós. No caso de aplicações de armazenamento de dados, pode ser necessário transferir grandes quantidades de dados. Isto porque os dados estão associados a um identificador e podem precisar ser transferidos caso o nó responsável por aquele identificador seja alterado. Nestes casos, a utilização de *informação de proximidade* entre nós pode influenciar favoravelmente o desempenho do algoritmo, uma vez que dados são transferidos por distâncias menores. Foram publicados alguns trabalhos que tratam da proximidade entre nós [112, 147]. Uma maneira de determinar distâncias entre nós de um sistema é utilizar o conceito de aglomeração por pontos de referência (*landmark clustering*) [112, 147]. Neste caso, cada nó determina sua distância a um determinado número de pontos de referência, que podem ser nós do próprio sistema, construindo um ve-

tor contendo estas distâncias. Se utilizarmos um número suficientemente grande de pontos de referência, podemos inferir que nós com vetores de distâncias similares estão localizados em pontos geográficos próximos.

5.2 Armazenamento distribuído de dados em redes *peer-to-peer*

Diversos sistemas de armazenamento distribuído de dados para redes *peer-to-peer* foram propostos recentemente [17,23,35,59,75,91,108]. A principal motivação para utilização das redes *peer-to-peer* são suas características desejáveis, como auto-organização, descentralização e escalabilidade. A maioria dos sistemas utilizam tabelas de espalhamento distribuídas para organizar os dados armazenados, mas alguns sistemas como o OceanStore [75] utilizam outras estruturas de dados, como as árvores de Plaxton [99]. Nesta seção, descreveremos alguns destes principais sistemas de armazenamento distribuído.

5.2.1 CFS

O *Cooperative File System (CFS)* [35] é um sistema de arquivos imutáveis baseado em uma tabela de espalhamento distribuída (DHT) implementada pelo Chord. O CFS permite o armazenamento e a recuperação de arquivos distribuídos de modo eficiente, robusto e tolerante a falhas. O seu funcionamento se baseia no armazenamento de blocos de dados em máquinas geograficamente distribuídas. A determinação do local de armazenamento dos blocos é feito através da DHT, onde o identificador dos blocos é gerado utilizando uma função de espalhamento segura, como a SHA-1, sobre o conteúdo do bloco. O acesso ao sistema de arquivos é feito por um cliente, que implementa a funcionalidade de um sistema de arquivos, e utiliza o CFS para armazenar os blocos deste sistema de arquivos.

Quebrar o arquivo em diversos fragmentos tem suas vantagens, já que isto automaticamente resulta em um balanceamento de carga, uma vez que fragmentos do arquivo serão recuperados a partir de várias máquinas diferentes. Por outro lado, o armazenamento e recuperação de um arquivo envolve a busca pelas máquinas responsáveis por armazenar cada um dos blocos que constituem o arquivo, aumentando o tempo de resposta para localizar e recuperar arquivos.

Para permitir tolerância a falhas, os blocos armazenados são replicados em diversas máquinas. Esta replicação também melhora o tempo de resposta do sistema, já que os blocos podem ser recuperados das máquinas mais próximas ou com conexões mais rápidas. Finalmente, o balanceamento de carga em máquinas heterogêneas é feito utilizando o conceito de servidores virtuais, onde o administrador de uma máquina pode manualmente determinar o número de servidores virtuais inicializados de acordo com a capacidade da máquina.

5.2.2 PAST

O PAST [108], assim como o CFS, é um sistema para armazenamento distribuído de arquivos imutáveis baseado em uma rede *peer-to-peer*. Mas o PAST se baseia no Pastry, ao invés do Chord, para implementar a funcionalidade de tabela de espalhamento distribuída. Outra diferença com o CFS é que o PAST armazena arquivos completos nos nós ao invés de blocos destes arquivos.

Para melhorar a disponibilidade dos arquivos, são armazenadas k réplicas dos arquivos. Quando um nó recebe uma requisição de armazenamento, ele escolhe outros $k - 1$ nós de seu conjunto de vizinhos para armazenar réplicas do arquivo. Quando um nó não possui espaço de armazenamento disponível para armazenar uma réplica de um arquivo, este repassa a requisição de armazenamento para outro nó de sua vizinhança. Deste modo, o mecanismo de balanceamento de carga do PAST não considera explicitamente a heterogeneidade dos nós.

5.2.3 OceanStore

O OceanStore [75] permite a criação de um sistema de escala global para o armazenamento persistente para dados mutáveis. Cada arquivo (ou objeto) armazenado possui um identificador global único (GUID). Para prover tolerância a falhas, dados são armazenados utilizando replicação.

A localização dos objetos é feita utilizando uma versão modificada dos *filtros de Bloom* [19]. Estes filtros permitem realizar testes de inclusão de um membro em um conjunto, com a lista de membros armazenada de forma bastante compacta. Em compensação, existe uma pequena possibilidade de o teste de inclusão retornar falsos positivos. No caso do OceanStore, cada nó guarda uma lista de filtros de Bloom, contendo a lista de objetos armazenados no próprio nó e em nós vizinhos. Caso o sistema não consiga localizar o objeto utilizando filtros de Bloom, o sistema utiliza uma segunda estrutura de dados, denominada *árvores de Plaxton* [99]. Nesta estrutura de dados, cada nó recebe um identificador aleatório e estes identificadores são utilizados para construir múltiplas árvores, cada uma contendo um nó diferente como sua raiz. A partir destas árvores, é possível localizar o nó responsável por um determinado identificador em $(\log n)$ passos, onde n é o número de nós do sistema.

A modificação de dados armazenados na presença de diversas réplicas é realizada utilizando *atualizações*, que são estruturas de dados contendo as modificações a serem realizadas nos objetos. Quando diversas modificações precisam ser aplicadas em um único objeto em uma determinada ordem, um protocolo de consenso Bizantino é executado entre os participantes, de modo a escolher em qual ordem aplicar as modificações. No momento em que uma aplicação deseja armazenar um arquivo de forma permanente, o arquivo é codificado em fragmentos redundantes utilizando códigos de rasura (*erasure codes*) [85, 96, 101]. Estes fragmentos são então distribuídos em centenas de servidores, garantindo sua durabilidade. Arquivos armazenados de forma permanente não podem ser modificados.

Finalmente, mecanismos de introspecção são utilizados para aglomerar arquivos relacionados e gerenciar as réplicas de arquivos. A partir deste mecanismo, o sistema automaticamente se otimiza para

permitir uma melhora no desempenho da recuperação de arquivos e na disponibilidade dos dados.

5.2.4 Outros sistemas de armazenamento *peer-to-peer*

Existem diversos outros sistemas de armazenamento distribuído de dados. Descreveremos brevemente alguns destes sistemas nesta seção.

Glacier [59] permite o armazenamento estável e distribuído de dados mesmo na presença de grandes falhas correlacionadas. Para tal, o sistema utiliza códigos de “rasura” para codificar o arquivo em fragmentos redundantes e armazená-los de modo distribuído. Glacier possui mecanismos para gerenciar os fragmentos armazenados, incluindo sua recuperação em caso de falhas e saídas de nós. Arquivos armazenados recebem *leases* [71] com duração limitada e que precisam ser renovados pelo dono do arquivo. Um mecanismo de coleta de lixo periodicamente elimina os fragmentos de arquivos com *leases* vencidos.

Total Recall [17] é um sistema de armazenamento *peer-to-peer* baseado em componentes do CFS cujo foco é no gerenciamento automático da disponibilidade dos dados armazenados. Para tal, o sistema se baseia na predição de disponibilidade das máquinas e no gerenciamento do grau de redundância dos arquivos armazenados.

Kosha [23] utiliza uma rede *peer-to-peer* para armazenar dados de modo distribuído, utilizando a semântica de sistema de arquivos do *Network File System (NFS)*. O objetivo é adicionar ao NFS as vantagens dos sistemas *peer-to-peer* como auto-organização, replicação de dados, transparência de localização e escalabilidade. A sua implementação é baseada no Pastry e utiliza replicação de arquivos para garantir a disponibilidade dos dados.

Ivy [91] é um sistema de arquivos *peer-to-peer* que permite a leitura e escrita de arquivos. O sistema é totalmente baseado em registros (*logs*), que são armazenados no sistema *peer-to-peer* utilizando uma tabela de espalhamento distribuída. O sistema mantém um registro por usuário, que modificam seus arquivos escrevendo neste registro.

5.3 Armazenamento de dados em grades computacionais

Aplicações que executam em grades computacionais muitas vezes utilizam e produzem grandes quantidades de dados. Nesta seção analisaremos algumas das abordagens existentes para o gerenciamento de dados de aplicações em grades computacionais.

5.3.1 Grades de dados

As grades de dados [31, 138] são projetadas para permitir a execução de aplicações científicas que requerem grandes quantidades de dados, da ordem de Terabytes ou até Petabytes. Este dados são normalmente compartilhados entre diversas organizações, universidades ou institutos de pesquisa.

Estas aplicações requerem que as grades de dados forneçam uma infra-estrutura que permita o armazenamento de dados, o transporte de dados entre diferentes sítios e a execução de aplicações em máquinas localizadas próximas dos dados. Toda esta infra-estrutura deve ser construída em máquinas e redes de alto desempenho. Além disso, uma vez que os recursos são compartilhados, é necessário um mecanismo de segurança que controle o acesso aos dados armazenados.

O desenvolvimento de grades de dados é bastante complexo e envolve pesquisas em diversas áreas como: técnicas para o transporte eficiente de dados entre diferentes sítios, replicação e armazenamento de dados, localização de réplicas, escalonamento de execução de aplicações, controle de acesso aos dados e representação semântica dos dados. Em nosso trabalho estamos mais interessados no armazenamento de dados, de modo que descreveremos alguns trabalhos sobre transporte de dados e gerenciamento de réplicas.

5.3.2 Transporte de dados

Diversos trabalhos tratam do transporte de dados em grades computacionais e grades de dados. O GridFTP [2] é um padrão que adiciona características de segurança e desempenho, importantes para aplicações de grades computacionais, ao protocolo FTP. No caso da segurança, o GridFTP permite a autenticação utilizando o *Globus Security Infrastructure* (GSI) [90] ou Kerberos [72] e a transmissão de dados criptografados. Para melhorar o desempenho, o padrão permite que parâmetros do protocolo de transmissão do FTP sejam otimizados caso a caso. O GridFTP permite ainda que múltiplas conexões paralelas sejam abertas simultaneamente entre cliente e servidor. Outro recurso importante, especialmente para arquivos muito grandes, é o suporte à transferência de arquivos fragmentados em diversas localidades. Finalmente, o GridFTP permite que transmissões interrompidas possam ser reiniciadas do ponto em que foram interrompidas.

Kangaroo [130] é um sistema de movimentação de dados para grades. O sistema armazena os dados gerados pela aplicação no disco local e realiza a transmissão de dados para um repositório remoto em múltiplos passos ou estágios. Em cada passo, o sistema armazena os dados na memória ou discos rígidos de nós intermediários entre os pontos de origem e destino da transmissão. Este nós intermediários funcionam como *buffers* de dados, de modo que a transmissão pode ser executada incrementalmente. Deste modo, lentidões na rede ou falhas temporárias de comunicação fim-a-fim são escondidas do ponto de vista de aplicações que utilizam o Kangaroo.

Stork [73] é um gerenciador de transferências de arquivos em grades. O sistema realiza o monitoramento e gerenciamento das transferências de dados, garantindo sua confiabilidade. Além disto, o sistema implementa um algoritmo para realizar o planejamento das transferências de dados, que inclui o escalonamento das transferências baseado na heterogeneidade dos recursos e a adaptação dinâmica dos protocolos de transferência de dados. Deste modo, o sistema melhora a eficiência das transmissões de dados, adaptando-as a diferentes condições da rede e evitando que recursos fiquem sobrecarregados.

5.3.3 Armazenamento e gerenciamento de réplicas

As principais motivações para a utilização de réplicas são a redução da latência de acesso aos dados e melhorias na escalabilidade, desempenho e tolerância a falhas do sistema.

Muitos dos cenários onde uma grade de dados pode ser aplicada possuem configurações estáticas e bem definidas. Por exemplo, o GDMB [111] é um projeto piloto para o gerenciamento de dados de uma grade baseado num modelo cliente-servidor, onde servidores publicam listas de arquivos a serem replicados e clientes solicitam cópias destes arquivos para armazenar. O sistema foi projetado para operar em bases de dados apenas para leitura, de modo que não há a preocupação de manter consistência entre as réplicas. Finalmente, as configurações de quais dados cada cliente irá replicar são feitas manualmente.

Além do armazenamento, é necessário realizar o gerenciamento de réplicas utilizando, por exemplo, um *serviço de localização de réplicas (SLR)* [30, 106]. O Giggie [30] é um arcabouço arquitetural para o desenvolvimento de serviços de localização de réplicas. Arquivos armazenados no sistema são referenciados por uma representação lógica, que é um identificador único para o arquivo. O objetivo do serviço é localizar uma ou mais réplicas do arquivo a partir de sua representação lógica. O serviço permite diversos servidores para armazenamento de réplicas e, em cada servidor, um *catálogo local de réplicas (CLR)* realiza o mapeamento da representação lógica dos dados para a localização física da réplica. Para determinar em qual CLR procurar por uma réplica, um componente denominado *índice de localização de réplicas (ILR)* indexa os catálogos locais. Os ILRs podem ser organizados de diferentes formas, resultando em diferentes características de desempenho. Pode-se configurar o número de ILRs, o grau de redundância dos índices, o particionamento do espaço de nomes lógicos, o particionamento de CLRs alocados a cada ILR, o método de compressão para os dados dos ILRs e o protocolo de atualização dos índices.

Os sistemas que utilizam réplicas acima citados são úteis para grades de dados, onde normalmente os dados são utilizados apenas para leitura e são armazenados em repositórios dedicados. Para localizar as réplicas, são criados índices que podem ser consultados por aplicações clientes. Além disso, muitos destes sistemas possuem diversos itens estáticos, como os diretórios de réplicas no caso do serviço de localização de réplicas e os servidores no caso do GDMB. Mas o uso de configurações estáticas não é apropriado para grades computacionais de grande escala compostas por conjuntos dinâmicos de repositórios de dados.

Para resolver este problema, Cai *et al.* [24] desenvolveram um serviço de localização de réplicas baseado numa tabela de espalhamento distribuída. Neste serviço, cada catálogo local de réplicas (CLR) passa a ser responsável também por indexar a localização das réplicas de um conjunto de arquivos. Os índices que representam os arquivos são instanciados em um espaço de identificadores lógicos e cada CLR é responsável por realizar o mapeamento de uma faixa destes identificadores. Deste modo, é possível criar um serviço de localização de réplicas que seja também auto-organizável, escalável e tolerante a falhas.

Ripeanu e Foster [106] propuseram um sistema descentralizado e adaptativo para localização de réplicas. Neste sistema, diversos *nós localizadores de réplicas (NLR)* realizam o mapeamento entre os nomes lógicos de arquivos e a localização de suas réplicas. Cada NLR possui conhecimento global aproximado sobre a localização das réplicas no sistema. Para tal, os NLRs utilizam filtros de Bloom [19] para realizar a compressão dos dados armazenados nos NLRs e nas mensagens de atualização de estado. Os NLRs são conectados por uma rede *peer-to-peer* sobreposta, mas a estrutura da rede não é especificada no trabalho.

Foram propostos diversos outros sistemas para gerenciamento de réplicas, que utilizam diferentes técnicas para determinar as políticas de replicação. Lamehamedi *et al.* [76] utilizam a topologia de conexões entre os locais de armazenamento das réplicas, como em anel ou árvore, para determinar a melhor estratégia de replicação. O cálculo é feito considerando os custos para leitura e escrita nas réplicas. Bell *et al.* [15] propuseram uma técnica que utiliza um modelo econômico baseado em leilões, onde provedores de recursos fazem ofertas e clientes fazem propostas. Lee e Weissman [78] apresentam uma arquitetura de replicação dinâmica para o sistema de grade Legion [58]. Nesta arquitetura, cada nó de armazenamento determina se novas réplicas devem ser criadas ou removidas avaliando se esta operação traria uma melhoria no desempenho, resultando num número de réplicas proporcional à popularidade do arquivo. Ranganathan *et al.* [102] propuseram um modelo para o gerenciamento descentralizado de réplicas em uma rede *peer-to-peer*. Neste modelo, que aborda sistemas dinâmicos onde nós possuem alta probabilidade de estar indisponíveis, cada nó utiliza informações locais para determinar quando é necessário criar novas réplicas para garantir a disponibilidade de um arquivo.

5.3.4 Armazenamento com redes *peer-to-peer*

JuxMem [9] implementa um serviço de compartilhamento de dados para aplicações de grade unindo os conceitos de memória compartilhada distribuída e redes *peer-to-peer*. O objetivo é fornecer uma abstração de memória compartilhada que permita que aplicações manipulem dados com transparência de localidade e garantia de persistência. O sistema organiza os nós da grade em uma federação de aglomerados, cada qual contendo um nó gerenciador do aglomerado, que se conectam por uma rede sobreposta *peer-to-peer* estruturada como uma tabela de espalhamento distribuída (DHT). Para cada bloco de dados armazenado é criado um *grupo de dados*, que contém todas as máquinas que armazenam cópias deste bloco de dados.

Para possibilitar que o sistema encontre locais de armazenamento disponíveis, os nós provedores de recursos publicam um anúncio do espaço disponibilizado para armazenamento de dados. Estes anúncios são armazenados na própria DHT, aplicando uma função de espalhamento sobre o espaço disponível no nó. Clientes que desejam encontrar um nó provedor de recursos podem então procurar por estes anúncios na DHT. JuxMem permite ainda que dados armazenados sejam modificados. Neste caso, todos os nós pertencentes ao grupo de dados do bloco a ser modificado são contatados para atualizar suas cópias.

5.4 Resumo

Neste capítulo, apresentamos conceitos relacionados a sistemas *peer-to-peer* e armazenamento distribuído de dados em grades computacionais. Descrevemos os conceitos de redes *peer-to-peer* em detalhes, incluindo suas classificações e problemas de balanceamento de carga. Em seguida, apresentamos diversos sistemas de armazenamento distribuído de dados baseados em redes *peer-to-peer*. Finalizamos apresentando o conceito de grades de dados e diversos sistemas de gerenciamento e armazenamento de dados em grades computacionais.

Utilizando os conceitos de redes *peer-to-peer* descritos neste capítulo, apresentaremos no próximo capítulo os identificadores virtuais, que desenvolvemos para permitir o balanceamento dinâmico de carga em tabelas de espalhamento distribuídas.

Capítulo 6

Identificadores Virtuais

Ao desenvolvermos um sistema de armazenamento distribuído de dados que utiliza recursos computacionais de grades oportunistas, devemos tratar da dinamicidade e heterogeneidade destes recursos. Como vimos no Capítulo 5, a organização dos nós em uma rede *peer-to-peer* permite que o sistema funcione em ambientes altamente dinâmicos. Vimos também que existem diversas abordagens para tratar da heterogeneidade dos nós e do balanceamento de carga. Mas estas abordagens possuem diversas limitações com relação à flexibilidade e/ou sobrecarga gerada.

Neste capítulo, introduzimos o conceito de identificadores virtuais, que estendem sistemas que implementam tabelas de espalhamento distribuídas (DHTs) de modo a permitir o balanceamento dinâmico de carga entre nós heterogêneos. Ao utilizar identificadores virtuais sobre uma DHT, cada nó passa a ter dois identificadores, aquele atribuído pela DHT e um identificador virtual. O objetivo é utilizar a infra-estrutura da DHT ao máximo, de modo que os identificadores virtuais possam ser alterados com um baixo custo computacional.

Escolhemos o Pastry [109] como base para desenvolvermos os protocolos para utilização dos identificadores virtuais. Pastry implementa a funcionalidade de uma tabela de espalhamento distribuída e possui uma implementação de código aberto em Java denominada FreePastry [121]. Esta implementação foi utilizada em diversos projetos, nas áreas de comunicação em grupo [27], armazenamento distribuído de arquivos [108], *caching* de páginas da Internet [63] e distribuição de conteúdos digitais [26]. O conceito de identificadores virtuais pode ser utilizado com outras DHTs, como Tapestry e Chord, mas é necessário alterar os protocolos de identificadores virtuais para tratar das diferenças entre as DHTs.

Iniciamos o capítulo com a discussão de trabalhos relacionados na área de balanceamento de carga em DHTs, seguida da descrição do protocolo Pastry. Introduzimos então, na Seção 6.3, o conceito de identificadores virtuais e, na Seção 6.4, os diversos protocolos necessários para sua implementação sobre Pastry. Finalmente, apresentamos a implementação de identificadores virtuais sobre Pastry na Seção 6.5.

6.1 Trabalhos relacionados

Grande parte dos trabalhos em balanceamento de carga em DHTs utiliza o conceito de servidores virtuais [35, 54, 68, 103, 118]. Explicamos o conceito de servidores virtuais na Seção 5.1.3. Nesta seção comparamos algumas destas abordagens com a de identificadores virtuais.

A primeira abordagem proposta consistia em cada nó instanciar $O(\log n)$ servidores virtuais, onde n é o número de nós no sistema [35, 118]. Esta abordagem é de fácil implementação, uma vez que os algoritmos da DHT não precisam ser modificados. Mas ela requer que os nós mantenham $O(\log n)$ vezes mais tabelas, além da necessidade de trocas de mensagens para manter estas tabelas. Com o uso de identificadores virtuais, esperamos realizar o balanceamento de carga com uma sobrecarga muito menor, mantendo apenas duas tabelas extras de tamanho fixo por nó.

Karger e Ruhl [68] propuseram uma variação da abordagem de servidores virtuais para Chord onde cada nó instancia apenas um entre $\log n$ servidores virtuais alocados para o nó. O algoritmo consiste em cada nó escolher o servidor virtual que possua a menor distância para o seu sucessor. A vantagem desta abordagem é que não existe a sobrecarga causada pelo grande número de servidores virtuais. Mas este algoritmo ainda tem a limitação de funcionar apenas para nós homogêneos, além de não permitir o balanceamento dinâmico de carga. Além disso, quando um nó entra na rede, outros $O(\log \log n/n)$ nós precisam alterar seu endereço, de modo que pode ser necessário movimentar grandes quantidades de dados.

O conceito de servidores virtuais foi também aplicado para realizar o balanceamento dinâmico de carga [54, 103]. Para realizar o balanceamento, nós com alta utilização transferem servidores virtuais para nós com pouca utilização. O que diferencia os algoritmos de balanceamento dinâmico é a forma de determinar quais nós devem trocar servidores virtuais. Um algoritmo simples é permitir que cada nó periodicamente escolha outro nó de modo aleatório, trocando servidores virtuais se um estiver com alta e o outro baixa utilização. Uma segunda possibilidade é utilizar diretórios que armazenam informações sobre a utilização dos nós do sistema. Deste modo, é possível fazer escolhas mais informadas sobre quais nós devem trocar servidores virtuais. O balanceamento de carga com servidores virtuais sofre com a sobrecarga de cada nó ter que controlar diversos servidores virtuais simultaneamente. A idéia de utilizar identificadores virtuais é evitar esta sobrecarga.

Shen e Xu [112] utilizam uma arquitetura de dois níveis para projetar um algoritmo aleatório para balanceamento de carga que seja ciente da localidade dos nós. Para tal, criam uma rede composta de nós de alta capacidade que mantêm informações sobre a utilização e localização de nós vizinhos de capacidade menor. A vantagem desta abordagem é que ela permite que a transferência de servidores virtuais seja feita entre nós geograficamente próximos. Mas também sofre os efeitos da sobrecarga extra causada pelo uso de servidores virtuais.

Utilizando uma abordagem sem servidores virtuais, Karger e Ruhl [68] propõem que um servidor poderia transferir parte do seu espaço de identificadores com seus vizinhos para realizar o balanceamento

de carga. Xu e Bhuyan [144] propuseram um algoritmo onde nós vizinhos trocam faixas de identificadores utilizando informações sobre o histórico de acesso de arquivos e sobre a heterogeneidade dos nós. De modo similar ao nosso trabalho, o tamanho da faixa de identificadores alocada a cada nó depende de sua capacidade e pode ser alterada de acordo com a evolução do sistema.

Em ambos os trabalhos, os nós dividem uma parte do espaço de identificadores com seus vizinhos, alterando os identificadores fornecidos pela tabela de espalhamento distribuída. Mas alterar o identificador de um nó é custoso, já que é necessário alterar, criar e destruir diversas tabelas, como as de roteamento. Normalmente executa-se os protocolos de saída do nó antigo e entrada do novo nó, ambos bastante custosos. Para cada um dos protocolos, mensagens precisam ser enviadas aos vizinhos e aos nós que possuem ou possuirão o nó que alterou seu identificador em suas tabelas de roteamento. Uma alternativa consiste em alterar o protocolo Pastry, de modo a incluir uma nova operação de mudança do identificador de um nó. Ainda seria necessário contatar os vizinhos e todos os nós que precisassem alterar suas tabelas de roteamento, mas menos mensagens seriam necessárias. Além disso, para ambas as alternativas, vizinhos lógicos do nó também alteram seus identificadores, gerando uma quantidade ainda maior de informações trocadas pela rede. Em comparação, como veremos neste capítulo, a troca de identificadores virtuais é de baixo custo, uma vez que é necessário atualizar poucas tabelas. Além disso, diversos espaços virtuais podem ser instanciados simultaneamente e os identificadores virtuais destes espaços alterados independentemente. Isto fornece muito mais flexibilidade à aplicação *peer-to-peer*.

6.2 Pastry

Utilizamos o Pastry [109], desenvolvido por Antony Rowstron da Microsoft Research Ltd. e Peter Druschel da Rice University, como base para os protocolos de identificadores virtuais. Nesta seção apresentaremos os algoritmos e estruturas de dados do Pastry em detalhes, permitindo assim a posterior descrição do algoritmo de identificadores virtuais.

O *Pastry* fornece uma infra-estrutura que implementa a funcionalidade de uma tabela de espalhamento distribuída (DHT). Entidades em Pastry, como nós, chaves e mensagens recebem identificadores de um espaço de 128 bits. Estes identificadores são convertidos em um número na base 2^b , onde b é um parâmetro que pode ser escolhido pela aplicação, que define a relação entre o tamanho das tabelas de roteamento e o número de passos necessários para encontrar o nó de destino. Os nós são organizados em um anel e cada nó possui um *conjunto de folhas*, que contém os seus vizinhos lógicos situados neste anel. Esta tabela possui tamanho fixo L definido pela aplicação.

O roteamento de mensagens é feito através da comparação de prefixos, como podemos observar na Figura 6.1. O roteamento da mensagem pode ser iniciado a partir de qualquer nó do sistema. A cada passo do roteamento a mensagem é enviada para um nó cujo identificador tenha um prefixo em comum com o identificador da mensagem pelo menos um dígito mais longo que o prefixo em comum com o nó anterior. Para permitir o roteamento, os nós mantêm uma *tabela de roteamento*, contendo endereços de

Figura 6.1: Roteamento de mensagem em Pastry.

outros nós com prefixos em comum com este nó. Esta tabela possui $\lceil \log_{2^b} n \rceil$ linhas, onde n é o número de nós no sistema, com $(2^b - 1)$ entradas em cada. Cada linha l , $0 \leq l < \lceil \log_{2^b} n \rceil$, da tabela contém os endereços dos nós que compartilham os primeiros l dígitos com o nó que contém a tabela e cujos dígitos $l + 1$ correspondem aos $(2^b - 1)$ valores que este pode assumir. Por exemplo, para $b = 2$ e $2^b = 4$, o nó com identificador 3023 possui 4 linhas, sendo que a segunda linha ($l=1$) contém nós com identificadores 3023, 31XX, 32XX, 33XX. O primeiro item corresponde ao próprio nó que possui a tabela, enquanto os demais endereços correspondem a nós que compartilham o mesmo prefixo 3 que o nó que possui a tabela, mas que diferem no segundo dígito. Os dígitos X podem assumir qualquer valor.

Finalmente, um problema importante com algoritmos de roteamento para DHTs é que muitas vezes os saltos das mensagens ocorrem entre nós bastante distantes geograficamente. Isto resulta em esperas muito longas na busca por um nó. Para minimizar este problema, nós do Pastry mantêm uma tabela adicional chamada de *conjunto de vizinhos*, que contém um conjunto de nós geograficamente próximos. Este conjunto de nós é utilizado ao popular a tabela de roteamento do nó, de modo que esta contenha preferencialmente nós que estejam geograficamente próximos. O conjunto de vizinhos tem tamanho fixo, definido pela aplicação.

6.2.1 Protocolos

O *protocolo de roteamento de mensagens* garante que uma mensagem enviada ao sistema Pastry será entregue ao nó com o identificador numericamente mais próximo ao identificador da mensagem. Pastry utiliza o roteamento por prefixos, de modo que a cada passo a mensagem é enviada a um nó logicamente mais próximo ao de destino. Durante o roteamento, os nós sempre verificam se o nó de

destino da mensagem está presente em seu conjunto de folhas e, caso o encontre, redireciona a mensagem diretamente a este nó. Este normalmente corresponde ao último passo do protocolo, uma vez que à medida que a mensagem se aproxima do nó de destino, ela termina por chegar a um vizinho lógico daquele nó, que o contém em seu conjunto de folhas.

Para um conjunto de n nós, Pastry permite encontrar o nó com o identificador numericamente mais próximo àquele da mensagem em $\log_{2^b} n$ passos. Como a tabela de roteamento possui tamanho aproximado de $\lceil \log_{2^b} n \rceil \times (2^b - 1)$, a escolha de b envolve um compromisso entre o tamanho das tabelas de roteamento, e conseqüentemente o tráfego de rede para manutenção destas tabelas, e o número de passos para encontrar o nó de destino.

Para permitir que o sistema seja auto-organizável, foram definidos *protocolos para a entrada e saída de nós*. Quando um nó entra no sistema, este precisa inicializar suas tabelas de roteamento, de folhas e de vizinhos. Para tal, Pastry supõe que o nó X que está entrando no sistema conhece um nó A geograficamente próximo. O nó X então pede a A para realizar o roteamento de uma mensagem com chave K . Esta mensagem é então encaminhada ao nó Z com identificador numericamente mais próximo de K . Os nós contactados durante o roteamento da mensagem enviam parte de suas tabelas a X , que por sua vez utiliza esta informação para construir suas tabelas.

Nós podem também deixar o sistema, normalmente sem aviso. Cada nó Pastry periodicamente checa se seus vizinhos lógicos imediatos podem ser contactados. Caso um nó detecte que um de seus vizinhos está inalcançável, ele inicia o protocolo de saída de nós, onde os vizinhos lógicos do nó que deixou o sistema atualizam suas tabelas de folhas para substituir este nó. As tabelas de roteamento são atualizadas apenas quando um nó tenta contatar outro nó para rotear uma mensagem e não consegue. Neste caso, o nó com a tabela de roteamento desatualizada contata outro nó da tabela de roteamento para solicitar um substituto para aquele o nó que deixou o sistema.

6.3 Identificadores virtuais

Os identificadores virtuais, que propomos nesta tese, são uma extensão ao Pastry que permite criar um espaço virtual de identificadores que pode ser dinamicamente ajustado de acordo com a capacidade de cada nó. Um identificador virtual é atribuído a cada nó do sistema *peer-to-peer*, além do identificador já atribuído pelo Pastry, que chamaremos de identificador Pastry. A faixa de valores que os identificadores virtuais podem assumir é a mesma dos identificadores Pastry. O objetivo é permitir que, neste espaço virtual, identificadores virtuais de nós possam ser facilmente alterados de modo que cada nó seja responsável sempre por uma faixa do espaço virtual proporcional à sua capacidade. Esta capacidade é uma métrica escalar que pode ser definida em função de diversos parâmetros, como a disponibilidade média do nó, sua capacidade de armazenamento, largura de banda e poder de processamento. Os identificadores virtuais de um nó precisam ser alterados quando nós ingressam e saem do sistema e quando nós alteram suas capacidades. Neste caso, como veremos, é possível ajustar os identificadores virtuais dos nós para

refletir a nova configuração dos nós do sistema.

Os identificadores virtuais podem ser aplicados a diferentes classes de aplicações *peer-to-peer*, mas estas precisam satisfazer algumas condições:

- Supomos que os identificadores dos dados que serão armazenados são gerados de forma aleatória e uniforme, por exemplo, aplicando uma função de espalhamento sobre os dados. Isto porque o balanceamento de carga atribui a cada nó uma faixa de identificadores de tamanho proporcional à sua capacidade.
- A métrica que caracteriza a capacidade do nó precisa ser relativamente estável. Mudanças devem ocorrer numa escala de tempo maior do que aquela necessária para executar o protocolo que altera o identificador virtual do nó. O mesmo se aplica com relação à taxa com que nós entram e saem do sistema (*churn*).

Mesmo após instanciar um espaço virtual, o espaço de identificadores do Pastry continua inalterado. Deste modo, as aplicações podem realizar a escolha dos nós de destino de mensagens tanto no espaço virtual como no espaço de identificadores Pastry. Além disso, múltiplos espaços virtuais podem ser instanciados simultaneamente, bastando atribuir múltiplos identificadores virtuais a cada nó. Neste caso, cada identificador virtual pode corresponder a uma métrica diferente de capacidade, como confiabilidade, disponibilidade, espaço em disco ou largura de banda da rede de um nó. Ao realizar o roteamento de uma mensagem na rede, o sistema pode escolher qual destas métricas utilizar, ou optar por não utilizar nenhuma métrica. Neste último caso, o identificador Pastry dos nós é utilizado.

6.3.1 Visão geral dos protocolos e tabelas

Para utilizarmos identificadores virtuais em um sistema *peer-to-peer*, devemos definir protocolos para o roteamento de mensagens e para a atribuição e modificação dos identificadores virtuais. Além disso, os nós precisam manter tabelas que permitam a execução destes protocolos. A Figura 6.2 o processo de roteamento de mensagens no espaço de identificadores virtuais.

Podemos ver na figura os dois espaços de identificadores, o do Pastry e o virtual, e 4 nós, representados por diferentes figuras geométricas. Cada nó possui dois identificadores, o do Pastry e o virtual, que possuem valores distintos. Apesar disso, vemos que a posição relativa dos nós nos dois espaços de identificadores é a mesma, isto é, em ambos os espaços os nós possuem os mesmos vizinhos lógicos.

Podemos ainda ver na figura que o roteamento de mensagens destinadas ao espaço virtual é realizado no espaço de identificadores Pastry, com exceção do último passo, onde ocorre a transição entre os espaços de identificadores. Esta transição é feita através de uma tabela adicional, chamada de *conjunto de folhas virtuais*, que mapeia o espaço de identificadores original coberto pelo conjunto de folhas do Pastry para o espaço virtual. Para tal, o conjunto de folhas virtuais contém uma lista de nós logicamente vizinhos no espaço virtual e é centralizada no identificador original do nó. Para que esta transição seja

Figura 6.2: Identificadores virtuais versus identificadores do Pastry.

sempre possível, este conjunto precisa conter nós suficientes para cobrir, no espaço virtual, toda a faixa de identificadores coberta pelos nós do conjunto de folhas do Pastry. O tamanho do conjunto de folhas virtuais é variável para cada nó, uma vez que os nós contidos nos conjuntos de folhas Pastry e virtuais são distintos e as faixas de identificadores cobertas por estes nós também. Mas como ambos os espaços de identificadores possuem o mesmo número total de nós, em média, o número de elementos nos dois conjuntos será similar.

Quando um nó se conecta à rede, o protocolo de ingresso do Pastry é executado, de modo que o nó ingressante recebe um identificador Pastry aleatório e inicializa todas as suas tabelas relativas ao Pastry. O nó ingressante então inicia o protocolo de ingresso virtual, de modo a obter um identificador virtual e ficar responsável por uma faixa de identificadores virtuais de extensão proporcional à sua capacidade. Este processo de divisão do espaço virtual, que denominamos *protocolo de partição do espaço virtual*, é realizado em conjunto com os vizinhos lógicos do nó ingressante, contidos no conjunto de folhas do protocolo Pastry. Este protocolo é executado também quando um nó deixa o sistema ou quando a capacidade de um nó altera-se acima de um determinado limiar. Deste modo, os nós do sistema são sempre responsáveis por uma faixa de identificadores aproximadamente proporcional à sua capacidade.

Finalmente, cada nó mantém ainda os identificadores virtuais de seus vizinhos lógicos numa tabela denominada *conjunto de vizinhos virtuais*. Esta tabela é preenchida durante a partição do espaço virtual entre os vizinhos e é mantida pelos nós para permitir uma maior robustez durante a execução dos protocolos de roteamento e manutenção do espaço virtual.

6.4 Protocolos sobre Pastry

Nesta seção, descrevemos em detalhes os protocolos que permitem a utilização de identificadores virtuais em uma aplicação Pastry. Iniciamos discutindo o protocolo de roteamento de mensagens e, em seguida, apresentamos os protocolos que permitem alterar os identificadores virtuais dos nós.

6.4.1 Protocolo de roteamento virtual

O protocolo de roteamento é responsável por garantir a entrega de uma mensagem para o nó responsável pelo identificador da mensagem no espaço virtual. O objetivo ao desenvolver este protocolo foi aproveitar ao máximo as tabelas de roteamento do Pastry. Deste modo, diminuimos a quantidade de tabelas necessária para manter o espaço virtual, minimizando a sobrecarga de utilizar identificadores virtuais e a complexidade de implementação do protocolo de roteamento.

No protocolo de roteamento virtual, a mensagem é encaminhada de nó em nó utilizando o protocolo de roteamento do Pastry. Mas em cada passo, os nós verificam se seu conjunto de folhas virtuais contém o nó responsável pelo identificador virtual da mensagem. Em caso negativo, o roteamento continua normalmente. Em caso positivo, o roteamento é interrompido e a mensagem é redirecionada diretamente ao nó de destino.

O protocolo acima funciona desde que os conjuntos de folhas virtuais estejam atualizados. Mas, como veremos na Seção 6.4.2, durante a atualização de identificadores virtuais de um grupo de nós, este conjunto pode ficar desatualizado por curtos períodos de tempo, o que pode levar uma mensagem para um nó diferente daquele de destino. Neste caso, o nó que recebeu a mensagem erroneamente pode procurar pelo nó de destino correto utilizando o conjunto de vizinhos virtuais, de modo a redirecionar a mensagem para este nó. O conjunto de vizinhos virtuais é também utilizado como um atalho durante o roteamento da mensagem, apesar de não ser este seu objetivo principal.

No protocolo de roteamento do Pastry, cada nó n_{atual} que recebe uma mensagem que está sendo roteada, utilizando sua tabela de roteamento e seu conjunto de folhas, determina para qual nó $n_{próximo}$ a mensagem deve ser enviada em seguida. Quando utilizamos o processo de roteamento virtual, antes da mensagem ser enviada a $n_{próximo}$, o nó n_{atual} verifica se suas tabelas de vizinhos virtuais e folhas virtuais contém o nó responsável pelo identificador da mensagem no espaço virtual. Caso este nó responsável seja encontrado, a mensagem é redirecionada a este nó. O protocolo de roteamento no espaço virtual é descrito no Algoritmo 1.

Este procedimento garante que o protocolo do Pastry será minimamente modificado e que a mensagem sempre chegará ao nó de destino. Isto porque o último passo do protocolo do Pastry ocorre quando um nó que está roteando a mensagem encontra o nó de destino em seu conjunto de folhas Pastry. Como veremos na Seção 6.4.2, o protocolo de partição do espaço virtual garante que conjunto de folhas virtuais cobre uma faixa de identificadores que contém aquela coberta pelo conjunto de folhas Pastry. Deste modo, o fato do algoritmo de roteamento do Pastry funcionar implica que o algoritmo de roteamento vir-

Algoritmo 1 Algoritmo de roteamento.

```

se nóAtual.éResponsável (mensagem.identificadorDestino) então
    nóAtual.processaMensagem (mensagem);
senão se nóAtual.vizinhosVirtuais.contémNóResponsável (mensagem.identificadorDestino) então
    nóResponsável  $\Leftarrow$  nóAtual.vizinhosVirtuais.nóResponsável (mensagem.identificadorDestino)
    mensagem.enviaPara (nóResponsável);
senão se nóAtual.folhasVirtuais.contémNóResponsável (mensagem.identificadorDestino) então
    nóResponsável  $\Leftarrow$  nóAtual.folhasVirtuais.nóResponsável (mensagem.identificadorDestino)
    mensagem.enviaPara (nóResponsável);
fim

```

tual também funciona. Além disso, o número de passos para encontrar o nó de destino de uma mensagem é o mesmo do protocolo Pastry, isto é, $O(\log n)$ passos, onde n é o número total de nós.

Como pudemos ver, o protocolo de roteamento de mensagens utiliza duas tabelas extras, o conjunto de folhas virtuais e o conjunto de vizinhos virtuais, ambas de tamanho médio constante. Ambos os conjuntos possuem um tamanho médio similar ao do conjunto de folhas do Pastry e precisam ser atualizados apenas quando o identificador virtual de um de seus membros mudam, o que ocorre quando o protocolo de partição do espaço de identificadores virtuais é executado. O número médio de nós contactados é constante em relação ao número de nós do sistema, uma vez que as tabelas que precisam ser atualizadas possuem tamanho médio constante.

O custo extra para manter um espaço virtual é baixo porque o protocolo de roteamento no espaço virtual utiliza toda a infra-estrutura do Pastry, que é mantida independentemente da existência do espaço de identificadores virtuais. Esta infra-estrutura inclui as tabelas de roteamento de tamanho dependente do número de nós no sistema, as tabelas necessárias para determinar os nós geograficamente próximos e as tabelas e protocolos de monitoramento e tolerância a falha do Pastry. Conseqüentemente, podemos criar múltiplos espaços virtuais simultaneamente com uma sobrecarga reduzida. Além disso, o espaço de identificadores do Pastry é mantido, permitindo o roteamento de mensagens também neste espaço.

6.4.2 Protocolo de partição do espaço virtual

O protocolo de partição do espaço virtual é executado como parte dos protocolos de ingresso e saída de nós e quando a capacidade de um nó é alterada acima de um determinado limiar. Nestes casos, é necessário que a faixa de identificadores virtuais designada ao nó e seus vizinhos seja atualizada. O protocolo de partição do espaço virtual realiza a divisão de uma faixa do espaço virtual entre um grupo de nós vizinhos.

A Figura 6.3 mostra os 4 passos do protocolo. Estes passos são descritos abaixo:

1. O nó p , que inicia o protocolo, envia uma mensagem a seus vizinhos requisitando o valor de suas capacidades. O número de vizinhos é configurável, podendo ser igual, por exemplo, ao número

Figura 6.3: Protocolo de partição do espaço virtual.

de nós no conjunto de folhas do Pastry, que costuma ter valores entre 8 e 32 nós. Na figura, os vizinhos são os nós $l1$, $l2$, $l3$, $r1$, $r2$ e $r3$, todos contidos no conjunto de folhas Pastry de p .

2. O nó p determina qual a faixa de identificadores virtuais coberta por seus nós vizinhos. Esta faixa de identificadores é então repartida entre o nó p e todos os vizinhos, atribuindo a cada nó uma faixa de identificadores de tamanho proporcional à sua capacidade. Os identificadores virtuais dos vizinhos mais extremos, $l3_{\text{virtual}}$ e $r3_{\text{virtual}}$, não são alterados, de modo a não afetar a distribuição de carga nos nós adjacentes àqueles participantes do protocolo. Como podemos ver na figura, os identificadores virtuais dos nós vizinhos a p são deslocados de modo a aumentar a faixa de identificadores virtuais atribuída a p .
3. Finalizada a partição do espaço virtual, o nó p envia aos seus vizinhos uma mensagem de atualização contendo a lista dos identificadores virtuais que foram alterados. Os vizinhos são os mesmos

contatados no primeiro passo do algoritmo. Estes vizinhos atualizam seu identificador virtual e conjunto de vizinhos virtuais e devolvem uma mensagem a p para confirmar o recebimento da mensagem de atualização. O nó p também atualiza seu identificador virtual e conjunto de vizinhos virtuais.

4. O nó p envia então a mensagem de atualização a todos os nós cujo conjunto de folhas virtuais contém nós que tiveram seus identificadores virtuais alterados. A determinação de quais nós precisam ter seu conjunto de folhas atualizado é simplificada pelo fato deste conjunto ser centralizado no identificador Pastry do nó. Deste modo, é enviada uma mensagem ao nó f responsável pelo identificador virtual do nó p no espaço de identificadores Pastry. Na figura, o conjunto de folhas virtuais de f contém os elementos $l1$, $l2$, p , $r1$ e $r2$. O nó f , por sua vez, repassa a mensagem para seus vizinhos lógicos, que por sua vez repassam para os seus vizinhos, até a mensagem atingir um nó cujo conjunto de folhas virtuais não precisa ser atualizado. Durante a atualização de seu conjunto de folhas virtuais, cada nó deve garantir que a faixa de identificadores coberta pelos nós do conjunto de folhas virtuais contenha a faixa de identificadores coberta pelo conjunto de folhas Pastry do nó.

Um ponto importante do algoritmo de partição do espaço virtual é que as operações de repartição do espaço de identificadores são realizadas apenas entre um subconjunto de tamanho fixo dos nós. Esta característica é essencial para permitir uma maior escalabilidade do sistema. Por outro lado, cada nó recebe uma faixa do espaço de identificadores particionando apenas uma fatia local do espaço virtual.

O protocolo exige a troca de mensagens entre o nó que iniciou o protocolo e seus vizinhos, para obter suas capacidades e informar seus novos identificadores virtuais. Além disso, é necessário notificar os nós cujos conjuntos de folhas virtuais contenham referências a nós que tiveram o seu identificador virtual alterado. Apesar do número exato de nós que cada conjunto de folhas virtuais possui ser variável, estes conjuntos, em média, possuem o mesmo tamanho L que o conjunto de folhas Pastry. Deste modo, o número de nós contactados durante o protocolo é $O(L)$, constante em relação ao tamanho do sistema.

6.4.3 Protocolo de ingresso de nós

O protocolo de ingresso de nós é executado sempre que um novo nó entra no sistema. Ele é iniciado pelo nó que entrou no sistema, após este já ter executado o protocolo de ingresso do Pastry e inicializado suas tabelas acopladas aos protocolos do Pastry.

O protocolo de ingresso de nós segue os mesmos passos do protocolo de partição do espaço virtual, com as seguintes diferenças: (1) o protocolo inclui o nó ingressante na partição do espaço virtual entre os vizinhos, atribuindo temporariamente como identificador virtual do novo nó o seu identificador Pastry, (2) o nó ingressante deve construir seus conjuntos de folhas e vizinhos virtuais a partir de informações dos nós vizinhos e (3) os demais nós devem incluir o novo nó em seus conjuntos de folhas virtuais.

Para a construção de seu conjunto de vizinhos virtuais, o nó ingressante utiliza as informações obtidas da partição do espaço virtual entre seus vizinhos. Já para construir o conjunto de folhas virtuais, o nó ingressante solicita aos seus vizinhos imediatos seus conjuntos de folhas virtuais. A partir das informações dos nós contidos nestes dois conjuntos, o nó ingressante pode construir o seu próprio conjunto de folhas virtuais. Esta solicitação é feita logo no começo do protocolo, no mesmo momento em que o nó requisita as capacidades de seus vizinhos. Finalmente, a inclusão do nó ingressante nos conjuntos de folhas virtuais dos demais nós é feita quando o nó ingressante envia as mensagens de atualização, contendo os novos identificadores virtuais.

O protocolo de ingresso requer uma troca de mensagens a mais que o protocolo de partição do espaço virtual, realizada quando o nó ingressante solicita a seus vizinhos imediatos seus conjuntos de folhas virtuais. Mas esta troca de mensagens envolve apenas o nó ingressante e seus vizinhos imediatos.

6.4.4 Protocolo de saída de nós

O protocolo de saída de nós é executado sempre que um nó s deixa o sistema, seja voluntariamente ou por motivo de falhas. Normalmente um nó deixa o sistema sem avisar os demais, de modo que as saídas de nós são normalmente detectadas através de um mecanismo de monitoramento. No caso do Pastry, cada nó é responsável por periodicamente verificar se seus vizinhos estão conectados. Caso um nó d detecte que um vizinho não está mais acessível, este inicia um protocolo interno de saída de nós do Pastry.

O protocolo de saída é iniciado pelo nó d que detectou que um vizinho estava inacessível, logo após o término do protocolo do Pastry, e é bastante similar ao protocolo de partição do espaço virtual. A principal diferença é que os nós que possuem referências a s precisam remover este nó de seus conjuntos de folhas e vizinhos virtuais.

6.4.5 Protocolo de atualização de capacidade

A capacidade de um nó pode mudar durante o tempo de vida do sistema. Dependendo da métrica utilizada para definir a capacidade, a amplitude desta variação pode ser bastante grande. Neste caso, para garantir a distribuição de carga, é necessário redistribuir o espaço virtual entre seus nós vizinhos. O protocolo de atualização de capacidade de um nó é iniciado pelo nó que teve sua capacidade alterada e segue o mesmo procedimento do protocolo de partição do espaço virtual.

As atualizações de capacidades requerem a troca de mensagens de controle e atualização de tabelas em diversos nós. Além disso, dependendo da aplicação que utilizar a rede *peer-to-peer*, pode ser necessário transferir outros dados entre os nós. Deste modo, para evitar transferências desnecessárias de dados, o sistema deve possuir um limiar de mudança na capacidade de um nó, que deve ser atingido para iniciar o protocolo de atualização de capacidades. Este limiar pode incluir um valor mínimo de alteração da capacidade, além de um tempo mínimo que a alteração na capacidade deve ficar acima deste limiar.

6.5 Implementação sobre o FreePastry

Fizemos a implementação de identificadores virtuais sobre o FreePastry [121]. Nesta seção, descrevemos detalhes sobre esta implementação, incluindo as principais classes envolvidas e os protocolos de roteamento e entrada e saída de nós. O leitor interessado pode obter mais detalhes sobre a implementação dos protocolos consultando diretamente o código fonte da implementação, disponível em <http://integrate.incubadora.fapesp.br/portal/software/OppStore>. Iniciamos a seção descrevendo o FreePastry para, em seguida, descrever a implementação de identificadores virtuais.

6.5.1 FreePastry

O FreePastry [121] é uma implementação de código aberto e escrita em Java do Pastry, realizada por membros de diversas universidades e instituições. FreePastry permite implementar aplicações que necessitam da funcionalidade de uma tabela de espalhamento distribuída (DHT). Uma aplicação que utilize o FreePastry deve instanciar um nó Pastry, representado pela classe `PastryNode`, e conectar este nó na rede *peer-to-peer* através da classe `PastryEndpoint`. Esta conexão é realizada fornecendo ao nó ingressante o endereço de algum nó que já esteja conectado na rede. Se não houver outros nós, o nó recém-criado pode iniciar uma nova rede.

FreePastry utiliza um modelo de programação orientado a eventos. Para que as aplicações possam interagir com a rede *peer-to-peer*, FreePastry define uma interface `Application`. Esta interface deve ser implementada por uma classe da aplicação e então registrada na classe `PastryEndpoint`, possibilitando a interação da aplicação com o sistema *peer-to-peer*. A interface possui três métodos: `forward`, `deliver` e `update`. O primeiro método é chamado pelo FreePastry durante o roteamento de mensagens, antes do nó reenviar a mensagem para outro nó da rede. Este método permite que a aplicação altere o conteúdo da mensagem ou o seu destino. O método `deliver` é chamado para notificar que aquele é o nó de destino de uma mensagem. O método `update` é chamado quando um dos vizinhos lógicos do nó entra ou deixa o sistema.

Aplicações podem definir diversos tipos de mensagens implementando a interface `Message`. Esta interface possui um único método virtual `getPriority`, que deve retornar a prioridade daquela mensagem. Para enviar uma mensagem, a aplicação chama o método `route` da classe `PastryEndpoint`, passando como parâmetros a mensagem e seu identificador, que será utilizado para realizar o roteamento para encontrar o nó de destino desta mensagem. Este método permite também que a aplicação forneça o endereço do nó de destino da mensagem, de modo que a mensagem seja enviada diretamente a este nó.

A troca de mensagens entre nós do sistema é feita de maneira assíncrona. Cada nó possui uma fila de mensagens e trata as mensagens recebidas em sequência. Uma *thread* periodicamente checa por mensagens na fila e, caso encontre alguma, analisa esta mensagem e a despacha para a aplicação através dos métodos `forward` e `deliver`. Após o término do processamento da mensagem por parte

Para permitir que aplicações utilizem o FreePastry com identificadores virtuais, definimos uma classe `VirtualNode`. A aplicação se registra na rede *peer-to-peer* através da classe `VirtualNode` ao invés da `PastryEndpoint`. Durante o registro, a aplicação deve fornecer também o valor da capacidade daquele nó e o espaço virtual em que está se registrando. A classe `VirtualNode` fornece também o método `routeMessage`, que recebe como parâmetros a mensagem a ser enviada e o identificador da mensagem. Este método realiza o roteamento da mensagem para o nó responsável pelo identificador fornecido. Além disso, o método `sendDirectMessage` permite que a aplicação envie uma mensagem diretamente a um determinado nó.

Múltiplos espaços virtuais podem coexistir simultaneamente. Definimos a classe `VirtualSpace` que contém todas as informações relacionadas a um espaço virtual do qual a aplicação deseje participar. Estas informações incluem o identificador virtual, a capacidade do nó de acordo com a métrica daquele espaço virtual, o conjunto de folhas virtuais, o conjunto de vizinhos virtuais e uma referência para aplicação registrada naquele espaço virtual. As mensagens daquele espaço virtual são redirecionadas a esta aplicação, de modo que, para a aplicação, a utilização de identificadores virtuais é transparente. É possível definir uma classe que implemente `Application` para cada espaço virtual. Deste modo, múltiplas aplicações podem ser registradas numa mesma infra-estrutura de roteamento Pastry. Na Figura 6.4, a classe `PeerToPeerApplication` representa uma aplicação *peer-to-peer* registrada em um espaço virtual.

Definimos também a classe `VirtualMessage`, que implementa a interface `Message`, mas contém alguns campos extras. Esta classe possui o atributo `virtualSpaceNumber`, que define em qual espaço virtual o sistema deve procurar o nó de destino da mensagem, sendo que o espaço virtual de número 0 (zero) corresponde ao espaço de identificadores do Pastry. Os atributos `messageTargetId` e `sourceNode` definem o identificador da mensagem e o nó que solicitou o roteamento da mensagem, respectivamente.

6.5.3 Conjunto de folhas e vizinhos virtuais

A classe `VirtualLeafSet` implementa o conjunto de folhas virtuais de um nó. Este conjunto contém uma lista ordenada de nós logicamente consecutivos no espaço virtual. Como vimos na Seção 6.4.1, a faixa de identificadores virtuais ($faixa_{folhasVirtuais}$) coberta pelos nós deste conjunto deve sobrepor a faixa de identificadores Pastry ($faixa_{folhasPastry}$) coberta pelos nós do conjunto de folhas Pastry. O número de nós em `VirtualLeafSet` é variável, dependendo da capacidade de seus nós e do tamanho de $faixa_{folhasPastry}$.

A operação realizada com mais frequência em `VirtualLeafSet` é a de busca pelo nó responsável por um determinado identificador virtual, implementada pelo método `getTargetNode`, que devolve o nó responsável pelo identificador fornecido ou `null`, caso o conjunto não contenha este nó. Para permitir que esta busca seja feita de modo eficiente, implementamos o conjunto como uma árvore ordenada.

Deste modo, é possível obter para um dado identificador virtual qual o nó imediatamente à sua esquerda (sentido anti-horário) e direita (sentido horário). Outras operações importantes são as de inserção e remoção de nós e atualização dos identificadores virtuais de nós presentes no conjunto.

A classe `VirtualNeighborSet` mantém o conjunto de vizinhos virtuais daquele nó, isto é, um conjunto de L nós logicamente localizados à esquerda do nó no espaço de identificadores virtuais e L nós logicamente à direita. Estes nós são os mesmos presentes no conjunto de folhas do Pastry, mas `VirtualNeighborSet` contém também os identificadores virtuais e capacidades destes nós. Optamos por replicar este conjunto para desacoplar a implementação dos identificadores virtuais do restante do Pastry. As operações sobre `VirtualNeighborSet` são as mesmas que as utilizadas em `VirtualLeafSet`, de modo que a implementação de ambos os conjuntos é similar.

6.5.4 Protocolos

Todas as mensagens que trafegam pela rede *peer-to-peer*, exceto as mensagens internas do Pastry, passam pelo método `forward` da classe `VirtualApplication` antes de serem entregues ou repassadas a outros nós. Este método analisa as mensagens recebidas pelo nó e determina se a mensagem está sendo roteada pela rede ou se é uma mensagem pertencente aos protocolos de partição do espaço virtual, ingresso de nós, saída de nós ou atualização de capacidades. No primeiro caso, a mensagem é repassada para a classe `VirtualMessageRouter`. No segundo caso, a classe `VirtualApplication` identifica a qual protocolo a mensagem pertence e chama os métodos adequados da classe que implementa o protocolo.

Protocolo de roteamento

O protocolo de roteamento, descrito na Seção 6.4.1, permite entregar uma mensagem para o nó responsável pelo identificador virtual desta mensagem no espaço virtual. Este protocolo é implementado pela classe `VirtualMessageRouter`. Antes de uma mensagem ser entregue ou reenviada a outro nó pelo protocolo Pastry, ela é repassada ao método `forward` da classe `VirtualMessageRouter`. Este método verifica se algum nó de seus conjuntos de folhas virtuais e de vizinhos virtuais é responsável pelo identificador virtual da mensagem. Se sim, a mensagem é redirecionada para ser entregue àquele nó. Caso contrário, a mensagem continua sua rota original.

Devemos lembrar que um nó sempre está contido em seu conjunto de vizinhos virtuais, implementado pela classe `VirtualNeighborSet`. Além disso, o protocolo de partição do espaço virtual sempre atualiza `VirtualNeighborSet` antes de `VirtualLeafSet`. Conseqüentemente, durante o roteamento, um nó sempre verifica a presença do nó de destino da mensagem em `VirtualNeighborSet` antes de `VirtualLeafSet`.

Protocolo de partição do espaço virtual

Para a implementação do protocolo de partição do espaço virtual, definido na Seção 6.4.2, criamos uma classe `VirtualPartitionProtocol` que realiza as tarefas do protocolo referentes ao nó que iniciou o protocolo. Ela possui três métodos principais:

- `startProtocol`: chamado para iniciar o protocolo. Obtém a lista dos vizinhos lógicos do nó a partir de seu conjunto de folhas Pastry e envia uma mensagem a estes vizinhos solicitando seu identificador virtual e capacidade atuais.
- `setNeighborInformation`: chamado quando o nó recebe uma resposta de um nó vizinho com sua capacidade e identificador virtual. O nó adiciona então uma entrada numa tabela interna com as informações recebidas. Após receber resposta de todos os vizinhos, o nó realiza a partição do espaço virtual. O identificador virtual do vizinho lógico mais à esquerda e o do mais à direita são utilizados como limites da faixa de identificadores particionada. Em seguida o nó atualiza seu `VirtualLeafSet` e `VirtualNeighborSet` e envia uma mensagem `VirtualUpdateMessage` contendo os novos identificadores virtuais a seus vizinhos lógicos para que estes também atualizem seus conjuntos.
- `setUpdateResponseReceived`: chamado quando o nó que iniciou o protocolo recebe a confirmação de que um nó vizinho atualizou suas tabelas com os novos identificadores virtuais. Após receber a confirmação de todos os nós, uma mensagem é enviada ao nó responsável pelo identificador virtual do nó que iniciou o protocolo no espaço de identificadores Pastry, de modo que este nó e seus vizinhos possam atualizar seus `VirtualLeafSet`. O protocolo é então finalizado.

A classe `VirtualApplication` realiza a tarefa de identificar as mensagens pertencentes ao protocolo de partição virtual e tratá-las ou despachá-las aos métodos das classes adequadas. As mensagens de atualização de identificadores virtuais, `VirtualUpdateMessage`, são entregues às classes `VirtualLeafSet` e `VirtualNeighborSet`, que atualizam seus elementos com os novos identificadores virtuais. As mensagens com respostas a requisições ou atualizações enviadas pelo nó que iniciou o protocolo aos nós vizinhos são repassadas à classe `VirtualPartitionProtocol`, que processa estas mensagens, dando prosseguimento ao protocolo.

O protocolo de partição do espaço virtual não é chamado diretamente, mas utilizado como parte de outros protocolos, como os de ingresso e saída de nós e atualização de capacidades. Como vimos na Seção 6.4, estes protocolos são bastante similares ao de partição, diferindo apenas em alguns detalhes. Conseqüentemente, optamos por implementá-los como especializações do protocolo de partição do espaço virtual, utilizando o padrão de projeto *template method* [50]. Além dos métodos concretos que implementam o protocolo de partição do espaço virtual, a classe `VirtualPartitionProtocol`

Figura 6.5: Identificadores virtuais.

possui quatro métodos virtuais que permitem que estes protocolos sejam implementados como especializações desta classe. Os métodos virtuais são:

- `protocolStarted`: chamado assim que o protocolo é iniciado.
- `collectedNeighborInformation`: chamado após todos os vizinhos lógicos do nó terem enviado seus identificadores virtuais e capacidades.
- `updateMessageReady`: chamado após o particionamento do espaço virtual, antes de enviar as mensagens com os identificadores virtuais atualizados para os vizinhos lógicos do nó.
- `protocolFinished`: chamado assim que o protocolo é finalizado.

A Figura 6.5 mostra as classes envolvidas com a implementação dos protocolos e os métodos virtuais. Estes métodos são normalmente utilizados para permitir que os protocolos atualizem o estado do nó com informações específicas daquele protocolo, modifiquem as mensagens enviadas ou até mesmo criem e enviem novas mensagens.

Protocolos de ingresso, saída e atualização de capacidades

O *protocolo de atualização de capacidades* é implementado por `VirtualUpdateProtocol`, uma subclasse de `VirtualPartitionProtocol`. Este protocolo realiza as mesmas tarefas que o de partição do espaço virtual, de modo que as implementações dos métodos virtuais são apenas para atualizar o nó local e definir as mensagens enviadas como sendo do protocolo de atualização de capacidades.

O *protocolo de ingresso de nós* é implementado pela classe `VirtualJoiningProtocol`. As tarefas extras executadas por este protocolo com relação ao de partição do espaço virtual são: (1) que o

nó ingressante precisa construir suas tabelas em `VirtualLeafSet` e `VirtualNeighborSet` e (2) que os demais nós precisam incluir o nó ingressante em suas tabelas.

Para a construção da tabela em `VirtualLeafSet`, o nó ingressante requisita, no interior do método `protocolStarted`, o conjunto de folhas virtuais de seus vizinhos imediatos. Ao receber as respostas, o nó constrói então o seu conjunto de folhas virtuais. A tabela de `VirtualNeighborSet` é construída utilizando informações locais providas da partição do espaço virtual, a partir do método `updateMessageReady`.

Já a inclusão do novo nó nos conjuntos de vizinhos virtuais e de folhas virtuais é feita pelas próprias classes `VirtualLeafSet` e `VirtualNeighborSet` no momento de atualizar seus elementos. Para tal, a mensagem de atualização de identificadores virtuais, `VirtualUpdateMessage`, contém um campo extra que identifica o nó que entrou no sistema.

O *protocolo de saída de nós* é implementado pela classe `VirtualDepartureProtocol`. Este protocolo é similar ao de atualização de capacidades, de modo que as implementações dos métodos virtuais apenas identificam o protocolo como de saída de nós. A principal diferença é que as mensagens `VirtualUpdateMessage` recebem agora um campo extra identificando o nó que deixou o sistema. Este nó é então removido dos conjuntos de folhas e de vizinhos virtuais dos nós que recebem esta mensagem.

6.6 Resumo

Neste capítulo apresentamos o conceito de identificadores virtuais, que permitem estender uma tabela de espalhamento distribuída para realizar o balanceamento de carga entre nós heterogêneos utilizando uma quantidade bastante reduzida de recursos. Apresentamos os principais protocolos necessários para manter estes identificadores virtuais e sua implementação sobre o FreePastry. No próximo capítulo apresentaremos um sistema de armazenamento distribuído para grades computacionais oportunistas que utiliza identificadores virtuais para tratar da heterogeneidade e dinamismo destas grades.

Capítulo 7

OppStore: Middleware para Armazenamento Distribuído de Dados

Neste capítulo apresentamos o projeto e implementação do OppStore [40, 41], um middleware para armazenamento distribuído de dados. OppStore utiliza máquinas não-dedicadas de grades oportunistas para armazenar dados de usuários e aplicações da grade. Estas máquinas são organizadas numa federação de aglomerados, que corresponde à organização de diversos sistemas de grades oportunistas, como o InteGrade [55], Condor [131] e OurGrid [33]. Estes aglomerados são conectados através de uma rede *peer-to-peer* estruturada como uma tabela de espalhamento distribuída. O OppStore codifica os arquivos que serão armazenados em fragmentos redundantes e distribui estes fragmentos em diferentes aglomerados. Utilizamos os identificadores virtuais para permitir que o sistema escolha, para armazenar os fragmentos, repositórios que permanecem ociosos por períodos de tempo mais longos, otimizando assim a disponibilidade dos fragmentos armazenados.

Iniciamos o capítulo apresentando trabalhos relacionados ao armazenamento distribuído de dados em grades computacionais e redes *peer-to-peer*. Em seguida, na Seção 7.2, apresentamos o projeto do middleware OppStore. Discutimos seus principais componentes na Seção 7.3, o gerenciamento de dados na Seção 7.4 e sua implantação em grades oportunistas na Seção 7.5. Discutimos então, na Seção 7.6, a implementação que fizemos do OppStore, discutindo o que já foi implementado e o que ainda está em fase de implementação. Finalmente, finalizamos o capítulo com um resumo e discussão do que foi apresentado.

7.1 Trabalhos Relacionados

O FreeLoader [137] possibilita a utilização do espaço livre em disco de estações de trabalho compartilhadas para o armazenamento de dados de aplicações científicas. Este sistema divide os dados em diversos fragmentos, que são armazenados em máquinas distintas, para melhorar o desempenho. De

modo similar a nosso trabalho, o FreeLoader também utiliza recursos ociosos de máquinas compartilhadas para armazenar dados de aplicações. Mas o FreeLoader considera apenas um pequeno conjunto estático de máquinas de um único aglomerado, enquanto nosso trabalho trata de conjuntos dinâmicos de milhares de máquinas distribuídas em múltiplos aglomerados. Além disso, o FreeLoader não considera a distribuição de carga e heterogeneidade das máquinas ao realizar o armazenamento de dados, como fazemos no OppStore.

O Stork [73] é um sistema que gerencia atividades de transferência de dados, de modo a torná-las mais confiáveis. O sistema esconde falhas de comunicação, trata da heterogeneidade de recursos e escala as transferências para evitar a sobrecarga de recursos. Já o Kangaroo [130] é um sistema de transferência de dados que tenta esconder a latência e limitações de banda da rede para aplicações que produzem arquivos de saída. Para tal, o sistema realiza a transferência em estágios, armazenando os dados em máquinas intermediárias. Diferentemente de nosso trabalho, o Kangaroo e o Stork tratam apenas de atividades relacionadas à transferência de dados, com o objetivo de melhorar sua velocidade e/ou confiabilidade. Deste modo, estes sistemas não possuem mecanismos para descoberta de repositórios e localização de dados armazenados.

O JuxMem [9] unifica os conceitos de redes *peer-to-peer* e memória compartilhada distribuída para implementar um serviço de compartilhamento de dados para aplicações da grade. Como em nosso trabalho, o JuxMem organiza as máquinas da grade em uma federação de aglomerados conectados por uma rede *peer-to-peer*. Mas o JuxMem implementa uma memória distribuída compartilhada e, para tal, precisa manter diversas réplicas completas dos dados armazenados. Isto causa uma sobrecarga muito maior no uso de redes de comunicação e espaço em disco, especialmente no caso de máquinas não-dedicadas, onde a redundância dos dados armazenados precisa ser maior. Além disso, o JuxMem não implementa mecanismos para balanceamento de carga e não utiliza informações sobre a capacidade das máquinas ao escolher os locais de armazenamento.

Uma técnica comum para grades de dados é o uso de replicação dos dados em conjunto com um sistema de localização de réplicas [24, 30, 106]. Giggie [30] é um arcabouço arquitetural para o desenvolvimento de serviços de localização de réplicas. O serviço permite diversos servidores para armazenamento de réplicas e as réplicas são indexadas por *índices de localização de réplicas (ILR)*. Estes índices são configurados de maneira estática e manual e supõe-se que os repositórios de dados são dedicados. Estas características não são adequadas para grades oportunistas, que são ambientes altamente dinâmicos formados por máquinas não-dedicadas. Para resolver o problema dos índices de réplicas estáticos, Cai *et al.* [24] desenvolveram um serviço de localização de réplicas baseado numa tabela de espalhamento distribuída, onde cada servidor de armazenamento de réplicas passa a ser também um servidor contendo *índices de localização de réplicas (ILR)*. Mas este sistema também supõe que os repositórios de réplicas são dedicados e não considera a heterogeneidade destes repositórios ao escolher o local de armazenamento das réplicas.

Ripeanu e Foster [106] propuseram um sistema de localização de réplicas descentralizado e adapta-

tivo. Este sistema é composto por diversos *nós localizadores de réplicas (NLR)*, onde cada NLR utiliza filtros de Bloom [19] para manter um conhecimento global aproximado sobre a localização das réplicas. Os NLRs são conectados por uma rede *peer-to-peer*, mas os autores não especificam a estrutura da rede e como as atualizações de estado são propagadas por esta rede. Diversos outros sistemas para gerenciamento de réplicas, que utilizam diferentes técnicas para determinar as políticas de replicação, foram propostos [15, 76, 78, 102]. Uma diferença importante entre nosso sistema para os que utilizam réplicas é a nossa ênfase no uso exclusivo de recursos ociosos de máquinas compartilhadas. Neste caso, para garantir a disponibilidade dos dados, codificamo-os em fragmentos redundantes e distribuímos-os por toda a grade, dando preferência para a escolha de repositórios com alta disponibilidade ao armazenar fragmentos. Além disso, nosso sistema permite que repositórios entrem e saiam do sistema constantemente, diversas vezes num mesmo dia, sem que seja necessário realizar reconfigurações ou transferências de dados significativas no sistema.

O OppStore possui similaridades com sistemas de arquivos distribuídos baseados em redes *peer-to-peer*, como o PAST [108] e o CFS (*Cooperative File System*) [35]. Ambos permitem armazenar arquivos em repositórios distribuídos e utilizam uma tabela de espalhamento distribuída para armazenar e localizar os arquivos. O PAST armazena réplicas completas dos arquivos, enquanto o CFS divide o arquivo em fragmentos, armazenando múltiplas cópias de cada fragmento. Mas em ambos os casos, a localização do arquivo ou fragmento fica vinculado ao seu identificador, de modo que quando repositórios alteram seus identificadores, os arquivos ou fragmentos precisam ser migrados, uma operação de custo bastante elevado. No OppStore, desvinculamos a localização de um fragmento de seu identificador, de modo que repositórios podem dinamicamente mudar seus identificadores sem o alto custo associado à migração de dados armazenados. Outro problema com o PAST e o CFS é relativo ao balanceamento de carga. No PAST, este suporte é restrito ao fornecido pelo uso de replicação dos dados. Já o CFS utiliza o conceito de servidores virtuais que, como já vimos, gera um grande aumento na sobrecarga de manutenção da rede *peer-to-peer*. No OppStore, utilizamos identificadores virtuais, que permitem o balanceamento dinâmico de carga entre nós heterogêneos com uma baixa sobrecarga de manutenção.

7.2 Projeto do middleware OppStore

No middleware OppStore, as máquinas são organizadas em uma federação de aglomerados, onde cada aglomerado é constituído por máquinas fisicamente próximas, por exemplo, em um mesmo laboratório ou departamento. Cada aglomerado deve conter uma máquina que instancia um módulo de gerenciamento dos repositórios de dados daquele aglomerado, denominado CDRM (*Cluster Data Repository Manager*). As demais máquinas funcionam como repositórios de dados e instanciam o módulo ADR (*Autonomous Data Repository*).

Os CDRMs formam uma rede sobreposta *peer-to-peer* estruturada, utilizando Pastry [109] como substrato. Cada CDRM instancia um nó Pastry e recebe um identificador de 160 bits, que é gerado

utilizando a função de espalhamento SHA-1 [44], de modo que o espaço de identificadores Pastry é particionado entre os CDRMs do sistema.

ADRs funcionam como repositórios de dados que recebem requisições para armazenamento e recuperação de dados em uma máquina. ADRs utilizam poucos recursos computacionais e podem ser configurados pelo dono da máquina para realizar transferência de arquivos somente quando a máquina está ociosa ou a qualquer momento.

Figura 7.1: Arquitetura do OppStore.

Na Figura 7.1 podemos visualizar a arquitetura do OppStore e seus principais componentes. O sistema é organizado em dois níveis: (1) rede *peer-to-peer* composta pelos CDRMs e (2) aglomerados compostos por um CDRM e vários ADRs cada. Esta organização em dois níveis facilita o gerenciamento do dinamismo da grade. Se todas as máquinas da grade fossem organizadas em uma rede *peer-to-peer* simples, as constantes mudanças de disponibilidade das máquinas, que normalmente ocorrem em ambientes de grades oportunistas, teriam que ser tratadas como operações de entrada e saída de nós. Utilizando este projeto em dois-níveis, este dinamismo pode ser tratado internamente em cada aglomerado pelo CDRM.

Para realizar o armazenamento de dados, OppStore codifica arquivos em fragmentos redundantes, utilizando uma codificação de rasura (*erasure coding*), e os distribui em diversos aglomerados da grade. Esta distribuição dos fragmentos melhora a disponibilidade do arquivo, uma vez que este pode ser recuperado mesmo quando ocorrem falhas em um ou mais aglomerados da grade. Além disso, durante a recuperação de arquivos, aplicações podem obter fragmentos localizados nos aglomerados mais próximos, melhorando o desempenho e diminuindo o tráfego de dados na rede. Finalmente, estudos analíticos [107, 141] mostram que, para um dado nível de redundância, dados armazenados com codificação de rasura possuem uma disponibilidade média várias vezes maior que utilizando replicação.

7.2.1 Armazenamento e recuperação de dados

Aplicações e usuários da grade acessam o OppStore através de uma biblioteca denominada *access broker*, que funciona como um intermediador de acesso ao sistema e esconde do usuário os detalhes dos protocolos utilizados internamente pelo OppStore. Esta biblioteca possui uma interface com métodos que permitem o armazenamento e recuperação de arquivos. Diversos tipos de dados podem ser armazenados num sistema de grade, com cada tipo de dados possuindo diferentes requisitos. OppStore permite que aplicações clientes escolham entre dois métodos de armazenamento para os arquivos: (1) armazenamento *perene* e (2) armazenamento *efêmero*.

Armazenamento *perene*

O *armazenamento perene* é utilizado para dados que serão armazenados por longos períodos. Neste modo, o arquivo é codificado em fragmentos redundantes que são distribuídos pelos aglomerados da grade. A Figura 7.2 mostra o processo de armazenamento *perene* de um arquivo na ausência de falhas, que é realizado em vários passos:

1. O *access broker* codifica o arquivo em fragmentos redundantes, utilizando um algoritmo de dispersão de informação (IDA) [37, 101]. Após o término da codificação, o *access broker* atribui um identificador a cada fragmento aplicando uma função de espalhamento segura de sentido único, a SHA-1 [44], sobre o conteúdo destes fragmentos.
2. O *access broker* envia a lista de identificadores de fragmentos ao CDRM de seu aglomerado. Este CDRM realiza o roteamento de uma mensagem para cada identificador na rede *peer-to-peer*, de modo a encontrar o CDRM responsável por cada identificador de fragmento.
3. Cada CDRM responsável pelo identificador de um fragmento seleciona um ADR de seu aglomerado e devolve o endereço deste ADR ao CDRM original. O CDRM original repassa esta lista de endereços de ADRs ao *access broker*.
4. O *access broker* transfere o conteúdo dos fragmentos para os ADRs. Os dados são transmitidos diretamente do *access broker* para os ADRs, sem trafegar pela rede *peer-to-peer*.
5. Após o término da transferência, o *access broker* notifica o CDRM de seu aglomerado. O CDRM cria uma estrutura, chamada de FFI (*File Fragment Index*), que contém a localização e identificador de cada fragmento. O FFI recebe como identificador o valor da aplicação da função SHA-1 sobre a lista de identificadores dos fragmentos. O CDRM do aglomerado realiza o roteamento do FFI para armazenamento remoto no CDRM responsável pelo identificador do FFI.

Diversos problemas podem ocorrer durante o armazenamento de um arquivo. (1) Durante o processo de escolha de um ADR de seu aglomerado, é possível que o CDRM não encontre nenhum ADR com

Figura 7.2: Armazenamento perene de arquivos no OppStore.

espaço em disco suficiente. Neste caso o CDRM repassa a requisição para o CDRM vizinho. (2) A mensagem solicitando o endereço de um ADR, ou sua resposta, podem se perder durante o roteamento. Neste caso, o broker receberá um número insuficiente de endereços ADRs e realizará o armazenamento de um número menor de fragmentos. (3) Podem ocorrer problemas durante a transmissão dos fragmentos para os ADRs. Caso um ou mais fragmentos não sejam transferidos corretamente, o ADR marca aqueles fragmentos como indisponíveis no FFI. Deste modo, o arquivo é armazenado, mas com um número menor de fragmentos. (4) O armazenamento do FFI do arquivo pode falhar. Neste caso, o *access broker* tenta repetir o procedimento por duas vezes e, caso não consiga armazenar o FFI, o *access broker* retorna uma mensagem de erro à aplicação que solicitou o armazenamento e solicita aos ADRs a remoção dos fragmentos que foram armazenados.

A recuperação de dados no OppStore é também realizada em vários passos:

1. O *access broker* requisita ao CDRM de seu aglomerado o FFI associado ao arquivo desejado, utilizando o identificador deste arquivo, fornecido por uma aplicação ou usuário. Este CDRM realiza então o roteamento de uma mensagem para o CDRM responsável por este identificador, que devolve o FFI correspondente. Caso o FFI do arquivo não seja localizado, o CDRM retorna uma mensagem de erro ao *access broker*, que por sua vez retorna um erro à aplicação que solicitou o arquivo.
2. Após receber o FFI, o *access broker* verifica sua integridade, aplicando a função SHA-1 sobre a lista de identificadores dos fragmentos e comparando com o identificador do arquivo. Em seguida, o *access broker* realiza a recuperação dos fragmentos, escolhendo para a transferência os repositórios mais próximos ou com maior velocidade de conexão. O *access broker* verifica então a integridade dos fragmentos, comparando o valor do SHA-1 sobre seu conteúdo com os identificadores destes fragmentos.
3. Caso consiga recuperar corretamente um número suficiente de fragmentos, o *access broker* reconstrói o arquivo a partir do conteúdo destes fragmentos. O arquivo é então devolvido à aplicação ou usuário. Caso contrário, o *access broker* devolve uma mensagem de erro. Os problemas que podem ocorrer são erros na verificação da integridade no conteúdo do FFI ou dos fragmentos e problemas na transferência de fragmentos a partir dos ADRs.

Devemos enfatizar que o conteúdo dos fragmentos não trafega pelos nós da rede *peer-to-peer* durante o roteamento de mensagens para encontrar os nós que irão armazená-los. O conteúdo destes fragmentos é transferido diretamente entre o *access broker* e os ADRs. Isto é importante pois arquivos podem ter tamanhos da ordem de Gigabytes e sua transferência através da rede *peer-to-peer* geraria um tráfego muito maior e poderia sobrecarregar os CDRMs.

Além disso, ao colocar a localização dos fragmentos no FFI, esta localização passa a ser independente dos identificadores dos CDRMs. Deste modo, quando o identificador do CDRM em cujo aglomerado o fragmento está armazenado muda, não é necessário mover fragmentos entre aglomerados. Já os FFIs são recuperados utilizando seu identificador, de modo que estas estruturas podem precisar ser copiadas no caso de alteração nos identificadores de CDRMs. Mas o seu tamanho é bastante reduzido quando comparado ao de fragmentos de arquivos.

Armazenamento efêmero

O *armazenamento efêmero* é utilizado para dados que precisam ser armazenados e recuperados rapidamente e que serão utilizados por curtos períodos de tempo, da ordem de alguns minutos. Neste método de armazenamento, o sistema armazena o arquivo no aglomerado de onde a requisição foi submetida e

utiliza replicação para prover tolerância a falhas. A desvantagem deste modo de armazenamento é que o sistema oferece menos garantias sobre a disponibilidade dos dados armazenados. Entre os tipos de dados que poderiam ser armazenados no modo efêmero estão arquivos temporários e *checkpoints* [39, 46] de aplicações.

Para realizar o armazenamento no modo efêmero, o *access broker* solicita ao CDRM de seu aglomerado uma lista de ADRs daquele aglomerado, onde réplicas do arquivo podem ser armazenadas. O *access broker* transmite então as réplicas para estes repositórios e notifica o CDRM de seu aglomerado, que cria um FFI, contendo os locais de cada réplica e o valor do SHA-1 das réplicas. O CDRM atribui como identificador do FFI o valor do SHA-1 das réplicas e realiza o armazenamento do FFI no CDRM remoto responsável pelo identificador do FFI. Mas o CDRM do aglomerado também guarda uma cópia local, de modo que o FFI estará prontamente disponível quando for solicitado a partir do aglomerado.

A recuperação de dados é similar, com o *access broker* requisitando ao CDRM do aglomerado o FFI, obtendo o conteúdo de uma das réplicas de um repositório e devolvendo o arquivo à aplicação solicitante.

7.2.2 Identificadores virtuais no OppStore

OppStore cria um espaço virtual de identificadores e atribui um identificador virtual a cada CDRM. A divisão do espaço virtual entre os CDRMs é realizada baseada na capacidade das máquinas presentes em cada aglomerado. Este espaço virtual é utilizado para selecionar os aglomerados que irão armazenar os fragmentos de arquivos. O objetivo é armazenar os fragmentos em máquinas com mais tempo ocioso e maior quantidade de espaço livre em disco.

No OppStore, definimos a capacidade de cada máquina como uma função de sua disponibilidade média e o espaço livre em disco. Esta métrica é configurável, mas utilizamos como padrão o valor $capacidade = disponibilidade * MultiplicadorDoDisco$, onde *MultiplicadorDoDisco* assume o valor 1 se o disco daquela máquina tiver capacidade acima de um limiar, por exemplo 10GB, e decresce linearmente até 0 se o espaço livre em disco for menor. Já *disponibilidade*, que assume valores entre 0 e 1, corresponde ao tempo médio em que a máquina permanece ociosa. Esta média deve ser calculada sobre o período de pelo menos alguns dias, de modo a evitar flutuações diárias no padrão de uso da máquina. Já a capacidade de um aglomerado é definida como a soma da capacidade de todas as máquinas de seu aglomerado. Deste modo, vemos que a faixa de identificadores atribuída a um CDRM é a soma das faixas que seriam alocadas a cada ADR. As equações para cálculo das capacidades são apresentadas abaixo:

$$Capacidade_{ADR} = disponibilidade\ média \times multiplicador\ do\ disco$$

$$Capacidade_{CDRM} = \sum^{ADRs} Capacidade_{ADR}$$

A seleção dos CDRMs responsáveis por armazenar os FFIs e a busca por FFIs durante a recuperação de arquivos são realizadas no espaço de identificadores Pastry, utilizando o identificador do FFI.

Portanto, a recuperação de FFIs é independente dos identificadores virtuais, de modo que o identificador virtual de um CDRM pode mudar sem que seja necessário mover FFIs entre CDRMs. Esta migração só precisará ocorrer quando CDRMs entram ou deixam o sistema, o que não deve ocorrer com frequência. Finalmente, fragmentos nunca precisam ser migrados, uma vez que estes são localizados pelos endereços de ADRs contidos nos FFIs.

Conseqüentemente, CDRMs podem alterar seus identificadores virtuais sem que seja necessário migrar dados armazenados no CDRM e em outras máquinas de seu aglomerado. CDRMs alteram seus identificadores virtuais quando, por exemplo, ADRs entram ou saem de seu aglomerado. Além disso, CDRMs podem entrar e sair do sistema com um baixo custo, uma vez que apenas parte dos FFIs dos vizinhos imediatos do CDRM que saiu ou entrou precisam ser migrados. Deste modo, a sobrecarga para que o OppStore se adapte dinamicamente a mudanças no ambiente é reduzida consideravelmente.

7.3 Principais componentes

Os três principais componentes do OppStore são: (1) o gerenciador de repositórios de dados do aglomerado (CDRM), (2) o repositório autônomo de dados (ADR) e (3) o intermediador de acesso (*access broker*). Nesta seção, descrevemos o projeto destes componentes em mais detalhes. Apresentaremos, na Seção 7.6, quais dessas funcionalidades já foram implementadas e os detalhes desta implementação.

7.3.1 Gerenciador de repositórios de dados do aglomerado (CDRM)

Os CDRMs são responsáveis pelo gerenciamento dos ADRs presentes em seu aglomerado. Para realizar este gerenciamento, CDRMs mantêm informações sobre todos os ADRs em seu aglomerado, incluindo sua capacidade, endereço de rede, estado, espaço livre em disco e lista de fragmentos armazenados.

Quando um CDRM recebe uma requisição para armazenamento de um fragmento durante o processo de armazenamento de um arquivo, ele escolhe um ADR para armazenar este fragmento e devolve seu endereço de rede. As restrições na escolha do ADR são que ele esteja disponível naquele momento e que tenha espaço em disco suficiente para armazenar o fragmento. A lista dos fragmentos armazenados nos ADRs é utilizada para reconstruir fragmentos que são perdidos no caso em que ADRs deixam o sistema. Para tal, esta lista deve incluir os identificadores dos FFIs (*File Fragment Indexes*) que contêm cada fragmento.

A lista de ADRs também é utilizada quando o CDRM recebe uma requisição para armazenamento efêmero de um arquivo. Neste caso, o CDRM retorna uma lista de ADRs que armazenarão as réplicas do arquivo. Estas réplicas são tratadas pelo CDRM do mesmo modo que fragmentos.

CDRMs são também responsáveis pelo armazenamento de FFIs. Como FFIs são pequenos, contendo apenas alguns endereços de ADRs e identificadores de fragmentos, os FFIs são armazenados diretamente

na máquina executando o CDRM.

Para prover tolerância a falhas, informações contidas em um CDRM são replicadas em outros r CDRMs vizinhos, onde r é um parâmetro configurável. Quando um CDRM falha, ele é reiniciado em outro nó do aglomerado utilizando a informação replicada nos vizinhos. Este processo é iniciado por um dos ADRs do aglomerado, quando este não consegue mais se comunicar com o CDRM de seu aglomerado. Este ADR contata um dos CDRMs vizinhos, obtém as informações sobre o CDRM que deixou o sistema e inicia um novo CDRM.

Poderíamos evitar esta comunicação do ADR com o CDRM de um aglomerado vizinho se, ao invés de replicar os dados de cada CDRM em seus CDRMs vizinhos, replicássemos estes dados nos ADRs de seus aglomerado. Mas, nesse caso, quando um aglomerado deixasse o sistema, os demais aglomerados não seriam mais capazes de recuperar os dados do CDRM daquele aglomerado. Trataremos da saída de aglomerados do sistema na Seção 7.4.3.

7.3.2 Repositório autônomo de dados (ADR)

ADRs são responsáveis por armazenar fragmentos codificados de arquivos. Para tal, ADRs possuem uma interface com métodos para o armazenamento de fragmentos e recuperação de fragmentos armazenados, permitindo que dados sejam transferidos entre ADRs e *access brokers*.

Quando um ADR entra no aglomerado, ele se registra com o CDRM daquele aglomerado, informando o seu endereço de rede e a quantidade de espaço livre em disco. Este endereço de rede é fornecido pelo CDRM a *access brokers* para que estes possam se conectar ao ADR. Diferentes protocolos podem ser utilizados para a conexão entre *access brokers* e ADRs. No caso de conexões TCP, o endereço pode ser um par (endereço IP, porta). Para ADRs localizados em máquinas atrás de NATs¹, este endereço poderia conter o endereço IP de uma máquina *proxy*, que seria então utilizado para estabelecer uma conexão TCP. Já o espaço livre em disco é utilizado pelo CDRM, em conjunto com a disponibilidade do ADR, para definir a capacidade da máquina. O CDRM inicialmente atribui ao ADR o valor médio da disponibilidade dos demais ADRs do aglomerado. Após coletar informações sobre a disponibilidade média da máquina por um determinado período, uma semana, por exemplo, o CDRM passa a atualizar a capacidade do ADR dinamicamente, utilizando a disponibilidade média da máquina.

ADRs podem estar em três estados diferentes: *ocioso*, *ocupado* e *indisponível*. ADRs são responsáveis por notificar o CDRM de seu aglomerado sobre seu estado, além do envio periódico de mensagens do tipo *keep alive*, que permitem que o CDRM detecte falhas nos ADRs. As máquinas contendo ADRs podem ser tanto compartilhadas como dedicadas. No caso de uma máquina compartilhada, seu dono pode configurá-la para transferir dados apenas quando ela está ociosa ou a qualquer momento. Neste último caso, o dono da máquina pode, visando manter sua qualidade de serviço, limitar a velocidade do

¹Máquinas atrás de NATs (*Network Address Translation*) recebem um endereço IP válido apenas na rede local onde estão localizadas. Deste modo, estas máquinas não podem receber conexões provenientes de máquinas de fora desta rede local.

envio de dados quando a máquina está ocupada. Máquinas dedicadas são sempre consideradas como ociosas e, conseqüentemente, possuem uma capacidade maior.

7.3.3 Intermediador de acesso (*Access Broker*)

O *access broker* é uma biblioteca que permite que aplicações e usuários da grade acessem os serviços de armazenamento de dados do OppStore. Esta biblioteca provê uma API em C contendo funções para realizar o armazenamento e recuperação de dados.

Como descrevemos na Seção 7.2.1, dados armazenados no OppStore são codificados em fragmentos redundantes utilizando o algoritmo de dispersão de informação (IDA) [37, 101]. IDA é um código de rasura (*erasure coding*) que permite codificar um arquivo U de tamanho n em $m + k$ fragmentos codificados de tamanho n/m , com a propriedade de que podemos recuperar U com apenas m dos $m + k$ fragmentos codificados. Outra propriedade útil de IDA é que a codificação é linear, permitindo a codificação e decodificação incremental dos fragmentos. Portanto, durante o armazenamento, podemos transferir os fragmentos paralelamente à sua codificação. Do mesmo modo, durante a recuperação, o arquivo pode ser reconstruído incrementalmente, à medida em que o *access broker* obtém o conteúdo dos fragmentos dos repositórios.

Para reconstruir um arquivo, o *access broker* precisa obter apenas um subconjunto dos fragmentos armazenados, de modo que ele pode escolher os repositórios mais rápidos para transferir os fragmentos. Para estimar a taxa média de transferência de dados a partir de um aglomerado, o *access broker* aplica um algoritmo de envelhecimento (*aging*) sobre as taxas de transferências obtidas na recuperação de fragmentos anteriores. Finalmente, o *access broker* aplica alguma aleatoriedade no momento de escolher os ADRs dos quais fragmentos serão obtidos, de modo a prevenir que apenas aglomerados de um pequeno grupo sejam selecionados.

Armazenamento e recuperação de arquivos

O armazenamento remoto de um arquivo pode ser realizado de modo síncrono, assíncrono e quasi-síncrono. No *modo síncrono*, a *thread* que realizou a chamada ao *access broker* permanece bloqueada até o término do processo completo de armazenamento. Deste modo, é garantido que quando a chamada ao método de armazenamento retorna, o arquivo já está armazenado no sistema. No *modo quasi-síncrono*, a *thread* que realizou a chamada permanece bloqueada até o término do processo de codificação dos fragmentos. Este modo garante que quando a chamada do método retorna, o processo de armazenamento não depende mais dos dados fornecidos pela aplicação para armazenamento, permitindo que a aplicação continue sua execução e modifique estes dados. A aplicação pode ainda fornecer uma função de *callback* para que o *access broker* notifique a aplicação quando o processo de armazenamento terminar. Já no *modo assíncrono*, a chamada ao método retorna imediatamente. Neste caso, a aplicação pode fornecer duas funções de *callback*, uma para o término da codificação e outra para o término do

armazenamento. Neste modo, a aplicação não deve modificar os dados fornecidos para armazenamento enquanto o processo de codificação não for finalizado.

A recuperação de um arquivo pode ser realizada de modo síncrono ou assíncrono. No primeiro modo, a chamada ao método de recuperação só retorna quando o arquivo solicitado tiver sido integralmente recuperado. No segundo modo a chamada retorna imediatamente e a aplicação pode fornecer uma função de *callback* para ser notificada quando o arquivo tiver sido totalmente recuperado.

Integridade e segurança dos dados

A verificação de integridade dos dados é realizada aplicando o algoritmo de espalhamento seguro SHA-1 sobre o conteúdo do arquivo e dos fragmentos, armazenando-os no FFI. O sistema também permite que o conteúdo dos fragmentos seja criptografado, mas neste caso a aplicação ou usuário do sistema precisa fornecer uma chave que o *access broker* utiliza para criptografar os dados.

O OppStore não provê uma infra-estrutura para gerenciamento de chaves para a criptografia de dados e o cliente é responsável por fornecer as chaves criptográficas. Utilizamos esta abordagem porque os ambientes de grades computacionais normalmente já possuem uma infra-estrutura de segurança, que pode ser utilizada para gerenciar estas chaves.

7.4 Gerenciamento de dados armazenados

Num sistema de armazenamento distribuído composto por até milhares de repositórios não-dedicados, o gerenciamento de dados armazenados é parte fundamental. O gerenciamento inclui diversas tarefas, como (1) permitir a localização dos componentes de cada arquivo armazenado, (2) garantir a disponibilidade do arquivo, (3) tratar da entrada e saída dinâmica de repositórios do sistema e (4) otimizar o desempenho na recuperação de arquivos armazenados.

7.4.1 Disponibilidade de dados

Ao analisar propriedades de disponibilidade de dados armazenados, temos que considerar dois cenários. O primeiro é o de indisponibilidades temporárias, que podem ser causadas pelos donos das máquinas reobtendo seus recursos computacionais, desligamentos temporários das máquinas ou desconexões da rede. O segundo são as constantes alterações no conjunto de membros da grade, que são causadas pelas saídas, permanentes ou por longos períodos, de ADRs ou de aglomerados, resultando na perda de fragmentos armazenados. Para evitar que arquivos sejam perdidos, definimos um mecanismo para reconstruir estes fragmentos.

Na prática, pode ser difícil diferenciar os dois cenários, uma vez que não temos como determinar se uma máquina que não está respondendo irá se reconectar novamente ao sistema em pouco tempo. Normalmente a diferenciação se dá por um limiar; por exemplo, se a máquina passa alguns dias sem

responder, podemos supor que ela não retornará em breve ao sistema. Já no caso de máquinas compartilhadas que estão temporariamente ocupadas e, por este motivo, não podem ser utilizadas para armazenar e recuperar fragmentos, podemos facilmente concluir que estamos no primeiro cenário. Nesta seção discutimos o primeiro cenário e deixamos para discutir o segundo na Seção 7.4.3.

Um fator importante que causa indisponibilidades temporárias em máquinas compartilhadas é que estas normalmente são configuradas por seus donos para enviar e receber dados apenas quando estiverem ociosas. Estas máquinas ficam indisponíveis simplesmente porque estão sendo utilizadas por um usuário local. Diversos experimentos de monitoramento do uso de máquinas [20, 43, 92] indicam que existem padrões de usos correlacionados nos tempos de ociosidade e funcionamento de máquinas de um mesmo laboratório ou departamento. Por exemplo, como era de se esperar, durante o período noturno e finais de semana, o tempo médio de ociosidade é maior do que durante o período diurno em dias de trabalho. Além disso, diferentes ambientes, como corporativo e universitário, parecem possuir diferentes padrões de uso.

Outro fator de indisponibilidade é o desligamento das máquinas por curtos períodos. Máquinas podem ser desligadas por problemas temporários de software ou hardware, durante o período noturno, para economizar energia, ou durante falhas no fornecimento de energia elétrica. Além disso, precisamos considerar as propriedades de conectividade fim-a-fim da rede. Medidas da conectividade entre máquinas na Internet provêm uma estimativa de que entre 1.5% a 3.3% do total de tentativas de conexão falham [36].

Indisponibilidades temporárias podem reduzir o número de máquinas disponíveis pela metade num período de poucas horas. O uso do algoritmo de identificadores virtuais reduz o problema colocando fragmentos em máquinas com maior disponibilidade. Além disso, caso a grade seja compostas por aglomerados geograficamente distribuídos, o protocolo de armazenamento de dados normalmente coloca fragmentos em máquinas geograficamente distantes, minimizando o efeito de indisponibilidades correlacionadas. Finalmente, gerando um maior número de fragmentos durante a codificação dos dados é possível obter melhores níveis de tolerância a falhas [107, 141].

7.4.2 Gerenciamento de arquivos

O sistema precisa gerenciar os arquivos armazenados, incluindo seus fragmentos. Este gerenciamento é realizado através dos FFIs, que contêm: (1) o identificador de cada fragmento, definido como o valor da aplicação da função de espalhamento segura SHA-1 sobre seu conteúdo, (2) o endereço do ADR que armazena cada fragmento e (3) uma cadeia de caracteres que identifica o arquivo. O identificador do fragmento é utilizado durante o processo de reconstrução do arquivo para verificar a integridade do conteúdo do fragmento.

O FFI é identificado pelo valor do SHA-1 sobre os identificadores dos fragmentos e cadeia de caracteres que identifica o arquivo. Não incluímos os endereços dos ADRs ao calcular o identificador do FFI porque a localização de cada fragmento pode mudar à medida em que ADRs deixam o sistema e frag-

mentos são reconstruídos. O identificador do FFI é gerado ao armazenar um arquivo e posteriormente utilizado para recuperar este arquivo. Este identificador possui a propriedade de ser utilizado tanto para recuperar o arquivo quanto para verificá-lo, de modo que podemos dizer que arquivos armazenados no OppStore são auto-verificáveis [142].

Arquivos armazenados podem também ser modificados, mas o sistema fornece suporte apenas à adição de dados no final do arquivo (*appending*). Caso um cliente deseje modificar o arquivo de outra maneira, ele deve remover o arquivo antigo e criar um novo. Para realizar a adição de dados, o *access broker* recupera o FFI referente ao arquivo, codifica os dados que serão adicionados em fragmentos, calcula e adiciona os identificadores destes fragmentos ao FFI, calcula um novo identificador para o FFI e substitui o antigo FFI pelo novo. Já os novos fragmentos são transferidos para os repositórios que mantêm os fragmentos originais e adicionados ao final destes fragmentos. Deste modo, apenas o FFI relativo ao arquivo modificado precisa ser substituído, os fragmentos armazenados são mantidos em seus locais originais.

Para gerenciar arquivos armazenados que não são mais utilizados, OppStore provê um certificado de empréstimo (*lease*), com um tempo limitado de validade, para cada arquivo armazenado. Estes certificados podem ser renovados pelos clientes que armazenaram o arquivo a qualquer momento. Durante o período de validade do certificado, o OppStore tenta garantir que aquele arquivo ficará disponível. Após o término da validade do certificado, o arquivo e os fragmentos são marcados como livres para remoção. Clientes podem tentar recuperar ou renovar a validade de arquivos com certificados vencidos, mas o sistema não oferece nenhuma garantia de que isso será possível.

7.4.3 Gerenciamento de fragmentos

Quando máquinas saem do sistema, os fragmentos armazenados nestas máquinas são perdidos. Para garantir que os arquivos que contêm estes fragmentos possam ser recuperados, o sistema deve reconstruir estes fragmentos. Cada CDRM é responsável por monitorar os ADRs de seu aglomerado. Quando um ADR deixa o sistema, o CDRM verifica a lista de fragmentos que estavam armazenados naquele ADR e os identificadores dos FFIs que contêm estes fragmentos. O CDRM envia então uma mensagem para cada FFI, contendo a lista dos fragmentos que devem ser marcados como perdidos. Cada CDRM também monitora seus CDRMs vizinhos, de modo que no caso de um aglomerado deixar o sistema, os fragmentos armazenados naquele aglomerado também são marcados como perdidos.

Quando o número de fragmentos perdidos de um FFI passar de um limiar, o CDRM responsável por aquele FFI inicia um procedimento de substituição destes fragmentos. Para reconstruir fragmentos perdidos, o CDRM precisa obter os fragmentos necessários para recuperar o arquivo original, o que faz da reconstrução de fragmentos uma operação cara. OppStore realiza esta reconstrução apenas no caso de arquivos cujo certificado de aluguel (*lease*) não esteja vencido.

Finalmente, quando o certificado de aluguel de um arquivo vence, o CDRM responsável pelo seu

FFI envia mensagens ao CDRMs cujos aglomerados contêm fragmentos daquele arquivo. Cada CDRM notifica o ADR responsável por aquele fragmento para marcá-lo como vencido. Deste modo, se o ADR precisa liberar espaço em disco, ele escolhe primeiro os fragmentos vencidos.

7.4.4 Otimização de desempenho

Para melhorar o desempenho das operações de recuperação de arquivos armazenados, OppStore armazena parte dos fragmentos necessários para reconstruir um arquivo no aglomerado a partir de onde foi solicitado seu armazenamento. Caso um cliente do mesmo aglomerado posteriormente solicite o arquivo, parte dos fragmentos necessários para reconstruí-lo poderão ser obtido do aglomerado local. Além disso, caso o arquivo seja solicitado a partir de outros aglomerado próximos, também será vantajoso utilizar estes fragmentos. Este armazenamento local de parte dos fragmentos pode ser desabilitado se o cliente souber que é pouco provável que os dados sejam utilizados naquele aglomerado ou em um aglomerado próximo.

Para realizar esta otimização, durante o processo de armazenamento, quando o CDRM fornece ao *access broker* a lista de ADRs remotos para armazenar os fragmentos, ele também provê uma lista de ADRs locais para armazenar os fragmentos. O *access broker* então transfere os fragmentos para estes repositórios do aglomerado local.

7.5 Implantação sobre grades computacionais

A implantação do OppStore em grades oportunistas requer que seja definida uma interface para que usuários e aplicações da grade possam interagir com os dados armazenados. Além disso, os componentes do OppStore devem poder ser instanciados em máquinas compartilhadas da grade oportunista.

Nesta seção, discutimos como poderíamos realizar a implantação do OppStore sobre grades computacionais oportunistas, incluindo os componentes que precisam ser implementados e disponibilizados. Na Seção 7.6, apresentaremos a versão preliminar da implementação que realizamos da interface do OppStore com o middleware InteGrade [55].

7.5.1 Interface com grades computacionais

O objetivo das interfaces com grades computacionais é permitir que aplicações e usuários da grade possam acessar o armazenamento distribuído fornecido pelo OppStore. Usuários e aplicações da grade, que chamaremos a partir de agora de clientes, devem poder realizar facilmente as tarefas de armazenamento de dados e sua recuperação.

No caso de usuários, eles devem poder manipular os dados armazenados utilizando comandos simples, como *oppstore*, *oppappend* e *oppremove*, ou através de uma interface gráfica. Esta interface com o usuário se comunica com o *access broker* para realizar a interação do usuário com o OppStore.

Aplicações precisam obter seus arquivos de entrada e armazenar arquivos de saída. A utilização dos repositórios remotos deve ocorrer de modo transparente, de preferência sem a necessidade de recompilar o código original da aplicação. A disponibilização dos arquivos de entrada e armazenamento dos de saída pode ser feita pelo próprio middleware de grade ou através de uma biblioteca que intercepte chamadas de sistema e converta estas chamadas para requisições ao OppStore. Esta biblioteca pode também realizar outras funções, como esconder da aplicação a latência de rede e falhas transientes de comunicação, fornecendo um ambiente de execução mais confiável à execução da aplicação.

Checkpoints gerados periodicamente são armazenados utilizando o modo efêmero. Mas alguns *checkpoints* podem periodicamente ser armazenados no modo perene, provendo uma proteção extra no caso de falha ou desconexão do aglomerado onde a aplicação é executada.

7.5.2 InteGrade

Como primeiro caso de uso, analisamos a implantação do OppStore sobre o middleware para grades oportunistas InteGrade [55], descrito em detalhes na Seção 3.2.1. O InteGrade organiza as máquinas compartilhadas em uma federação de aglomerados, como podemos ver na Figura 7.3. Esta figura mostra a distribuição de componentes do InteGrade, como o GRM, AR, EM e LRM, descritos na Seção 3.2.1, e dos componentes do OppStore já implantados sobre o InteGrade.

Figura 7.3: Implantação do OppStore sobre o InteGrade.

Como podemos notar na Figura 7.3, ADRs são instanciados nas máquinas provedoras de recursos, de modo que as mesmas máquinas que disponibilizam seus ciclos computacionais passam a disponibilizar também o espaço livre em disco. Além disso, CDRMs são instanciados nas máquinas gerenciadoras do aglomerado que, apesar de não serem dedicadas, permitem que os módulos de gerenciamento funcionem mesmo que a máquina não esteja ociosa.

Outro ponto importante da implantação é o modo como usuários e aplicações da grade interagem com o OppStore. No modelo de submissão atual do InteGrade, usuários utilizam a interface gráfica ASCT (*Application Submission and Control Tool*) para armazenar arquivos executáveis de aplicações no módulo AR (*Application Repository*) e submeter tarefas para execução na grade. As tarefas são repassadas para execução nos LRMs (*Local Resource Managers*), que obtêm os binários da aplicação no AR e os arquivos de entrada a partir do ASCT. Quando a aplicação finaliza, os arquivos de saída da aplicações são enviados para o ASCT de onde a execução foi requisitada.

Este esquema traz diversas limitações à execução de aplicações. A primeira é que o ASCT precisa ficar conectado ao sistema por todo o processo de submissão, escalonamento, execução e obtenção de resultados, uma vez que o ASCT interage diretamente com o LRM. Além disso, não é permitido ao usuário da grade visualizar os resultados da execução em máquinas diferentes da utilizada para a submissão da execução. Finalmente, não é possível compartilhar dados entre diferentes aplicações em uma grade computacional, uma vez que para cada execução os arquivos de entrada e saída são obtidos e enviados diretamente ao ASCT.

Utilizando o OppStore, os arquivos de entrada e saída são armazenados em repositórios distribuídos. No momento de submeter uma aplicação para execução, os arquivos de entrada são enviados para armazenamento remoto. Quando o LRM recebe a requisição para execução, ele obtém os arquivos de entrada da aplicação no OppStore, e realiza a execução da aplicação. Após seu término, os arquivos de saída são armazenados no OppStore. Usuários da grade podem então obter estes dados armazenados de qualquer máquina utilizando a ferramenta gráfica ASCT. Finalmente, como dados de entrada e saída de uma aplicação ficam armazenados no OppStore, outras aplicações podem facilmente utilizar estes arquivos.

Outro uso do OppStore seria para o armazenamento de *checkpoints* de aplicações. O InteGrade possui um mecanismo próprio para realizar o armazenamento de *checkpoints* no aglomerado onde a aplicação é executada. Com o OppStore, o mecanismo de *checkpointing* passaria a utilizar o armazenamento perene do OppStore para parte dos *checkpoints* gerados, de modo a fornecer mais garantias sobre a disponibilidade destes *checkpoints*.

7.5.3 Condor

Condor [131] é um middleware para grades oportunistas que organiza as máquinas como uma coleção de aglomerados, chamados de *Condor pools*. Máquinas compartilham seus recursos executando um módulo denominado *resource daemon*, equivalente ao LRM do InteGrade. A alocação de tarefas a recursos é realizada pelo módulo *matchmaker*, que possui um registro de todos os *resource daemons* presentes em seu aglomerado. Os usuários da grade realizam a submissão de tarefas para execução utilizando o módulo *agent*.

Em uma grade Condor composta por diversos aglomerados, podemos implantar o OppStore instanciando CDRMs nas mesmas máquinas que executam os *matchmakers* e os ADRs nas máquinas que exe-

cutam os *resource daemons*. Conseqüentemente, as mesmas máquinas que compartilham ciclos ociosos passam também a compartilhar espaço livre em disco.

Condor permite que aplicações acessem remotamente a máquina de onde a requisição de execução foi realizada, para obter dados lá armazenados e transferir arquivos de saída. Este acesso remoto a dados requer que a aplicação seja religada a uma biblioteca que intercepta chamadas ao sistema de arquivos. Utilizando o OppStore, uma maneira de permitir que aplicações acessem o repositório distribuído de modo transparente é modificar esta biblioteca para obter e armazenar dados no OppStore. Poderíamos também modificar o *agent* para que este automaticamente solicite o armazenamento dos dados que serão utilizados pela aplicação no OppStore e obtenha os dados de saída da aplicação armazenados no OppStore. Finalmente, poderíamos disponibilizar uma ferramenta texto ou gráfica que permita aos usuários da grade manipular e visualizar os dados armazenados no OppStore.

7.6 Implementação

Realizamos a implementação do OppStore, incluindo seus três principais componentes (CDRM, ADR e *access broker*), o gerenciamento de dados e a interface para sua utilização com o InteGrade. Já implementamos as principais funcionalidades descritas neste capítulo, mas algumas ainda estão em fase de implementação. Nesta seção discutimos detalhes da implementação que realizamos destes componentes e funcionalidades e do que ainda precisamos implementar. Leitores interessados podem obter o código fonte do OppStore na página do projeto na Internet (<http://www.integrate.org.br/portal/software/OppStore>).

7.6.1 Principais componentes

Implementamos os três principais componentes do OppStore: CDRM, ADR e *access broker*.

CDRM

O CDRM utiliza dois métodos de comunicação: (1) para se comunicar com CDRMs de outros aglomerados, o CDRM utiliza o protocolo de roteamento de Pastry e (2) para se comunicar com ADRs e *access brokers* de seu aglomerado, o CDRM utiliza o CORBA. Implementamos o CDRM em Java utilizando o JacORB [124], uma implementação de código aberto em Java da especificação CORBA, e uma versão do FreePastry estendida com identificadores virtuais, descrita na Seção 6.5. O diagrama UML na Figura 7.4 mostra as principais classes que fazem parte do CDRM.

Para realizar a comunicação entre CDRMs, definimos a classe *CdrnMessage*, especialização de *VirtualMessage*, que representa as mensagens correspondentes a operações do OppStore, como armazenamento e recuperação de arquivos. A classe *StoreFragmentMessage* representa mensagens

Figura 7.4: Diagrama de sequência do processo de armazenamento de arquivos.

enviadas por um CDRM para requisitar o endereço de um ADR para o armazenamento de um fragmento. Já a classe `FileFragmentIndexMessage` representa uma solicitação para recuperação ou armazenamento de um FFI.

A classe `CdrmApp`, que implementa a interface `Application` definida no `FreePastry`, recebe e processa as mensagens recebidas pelo CDRM providas, através do protocolo `Pastry`, de CDRMs de outros aglomerados. As mensagens podem ser de dois tipos: (1) requisição do endereço de um ADR para armazenamento de um fragmento, que é repassada para a classe `AdrManagerImpl`, responsável por gerenciar os ADRs do aglomerado e (2) requisição para o armazenamento de um FFI no CDRM, que é repassada para a classe `FileFragmentIndexManager`, responsável pelo gerenciamento de FFIs armazenados naquele CDRM.

Definimos duas interfaces CORBA para o CDRM. A primeira interface, `CdrmRequestManager`, é utilizada pelo *access broker* para submeter requisições para o armazenamento e recuperação de dados armazenados remotamente. A segunda interface, denominada `AdrManager`, é utilizada pelos ADRs

e permite que estes se registrem no CDRM, atualizem seu estado e enviem mensagens periódicas de atualização de seu estado.

A classe `CdrMgrImpl` implementa a interface CORBA `CdrMgr` e é responsável por processar as requisições de armazenamento e recuperação de arquivos. Ao receber as requisições, esta classe cria mensagens correspondentes à requisição e repassa-as à classe `VirtualNode`, que realiza o roteamento da mensagem a outros CDRMs. Ao receber as respostas destas mensagens, `CdrMgrImpl` envia os resultados obtidos ao *access broker* que realizou a requisição. Descreveremos esta interface e a implementação de suas operações em detalhes na Seção 7.6.2.

Já a classe `AdrMgrImpl`, que implementa a interface CORBA `AdrMgr`, realiza o gerenciamento dos ADRs presentes no aglomerado. A interface `AdrMgr` contém operações que permitem que ADRs se registrem, enviem mensagem periódicas de atualização de seu estado, notifiquem o recebimento de um fragmento para armazenamento de um *access broker* e obtenham listas de fragmentos que podem ser removidos. O código da interface é listado abaixo:

```
1 interface AdrMgr {
2     long registerAdr( in string address, in long freeStorageSpace );
3     void setFragmentStored ( in long adrId, in long fragmentSize );
4     long updateAdrStatus ( in long adrId, in long freeStorageSpace,
5                           in boolean isAvailable );
6     keySequence getFragmentRemovalList( in long adrId );
7 };
```

A classe `AdrMgrImpl` mantém uma lista com as características dos ADRs de seu aglomerado e é responsável por realizar a seleção de ADRs que armazenarão fragmentos de arquivos. A classe `AdrMgrImpl` seleciona apenas ADRs que estejam disponíveis no momento da requisição e contenham espaço livre em disco suficiente para armazenar o fragmento. Além disso, a *thread* implementada por `AdrLivenessMonitor` periodicamente verifica se existem ADRs que não enviam mensagens de atualização por muito tempo, removendo-os da lista.

A implementação atual do CDRM possui algumas limitações. Ainda não implementamos o suporte a tolerância a falhas dos CDRMs. Para tal, seria necessário implementar a replicação dos dados contidos no CDRM em seus CDRMs vizinhos. O envio de dados entre os CDRMs poderia ser realizado utilizando mensagens Pastry. Além disso, a versão atual do ADR não realiza o procedimento de reinicialização do CDRM quando ele detecta que o CDRM não está respondendo. Para tal, os CDRMs precisariam prover um novo método em sua interface CORBA que permita ao ADR obter as informações necessárias para reiniciar o CDRM que falhou.

ADR

Ao implementar o ADR, precisamos restringir o seu uso de memória, uma vez que o ADR é executado permanentemente nas máquinas provedoras de recursos. Para tal, utilizamos a linguagem C++,

uma vez que esta, além do alto desempenho, realiza menor uso de memória que linguagens que utilizam máquinas virtuais, como Java.

Assim que o ADR é iniciado, ele se registra com o CDRM de seu aglomerado, enviando suas características, como estado atual e espaço livre em disco na máquina. Além disso, uma *thread* periodicamente envia atualizações ao CDRM para informar que aquele ADR está ativo. A comunicação com o CDRM é feita utilizando o OiL [128], uma implementação código aberto em Lua da especificação CORBA. Optamos por utilizar o OiL devido ao seu baixo consumo de memória.

O ADR instancia um servidor que espera por conexões para transmissão e recepção de fragmentos a partir do *access broker*. Para tal, este servidor abre um soquete TCP/IP numa porta da máquina e aguarda por conexões provindas do *access broker*. Utilizamos soquetes ao invés de CORBA na transferência do conteúdo dos fragmentos para obter melhor desempenho. Para cada *access broker* que se conecta ao ADR, um novo processo é iniciado para tratar desta conexão, de modo que múltiplas transferências podem ocorrer simultaneamente. A transferência do conteúdo dos fragmentos é realizada utilizando o protocolo TCP e os dados são armazenados no disco da máquina à medida que são recebidos.

Access broker

Implementamos o *access broker* em C++ como uma biblioteca compartilhada. Definimos uma interface em C com funções para realizar o armazenamento e recuperação de dados. Definimos funções para realizar o armazenamento de modo síncrono, assíncrono e quasi-síncrono e para realizar a recuperação de modo síncrono e assíncrono. Utilizamos o C++ na implementação porque, no caso do *access broker*, nossa principal preocupação é com o desempenho do processo de codificação dos fragmentos.

O *access broker* precisa se comunicar com o CDRM de seu aglomerado. Esta comunicação é realizada para submeter requisições para obtenção de uma lista de endereços de ADRs para armazenar fragmentos, requisições para o armazenamento de FFIs e requisições para a obtenção de FFIs. Neste caso, o *access broker* invoca operações da interface CORBA definida no CDRM. O CDRM processa as requisições de modo assíncrono e as respostas são enviadas posteriormente ao *access broker*. Para tal, o *access broker* instancia um servidor CORBA que permite que o CDRM lhe envie respostas às requisições feitas. Do mesmo modo que nos ADRs, utilizamos o ORB OiL, escrito em Lua, para realizar as comunicações em CORBA. Listamos abaixo a interface CORBA do *access broker*:

Para codificar os arquivos em fragmentos redundantes, utilizamos uma versão otimizada do algoritmo de dispersão de informação (IDA), proposta por Malluhi e Johnston [87], que reduz a complexidade computacional do processo de codificação e decodificação. Fizemos uma descrição detalhada deste algoritmo de codificação na Seção 2.4.1.

Para produzir o identificador do FFI e dos fragmentos utilizamos a implementação da função de espalhamento segura (SHA-1) da biblioteca openssl [129]. Ainda não implementamos o suporte à encriptação dos dados que serão armazenados, que será realizada utilizando funções disponíveis na biblioteca

openSSL.

Finalmente, ainda não implementamos a estimativa da largura de banda entre o *access broker* e os aglomerados da grade para permitir a seleção dos ADRs a partir dos quais os fragmentos serão recuperados. Na versão atual, o *access broker* seleciona aleatoriamente os ADRs que serão utilizados para recuperar os fragmentos, de forma a distribuir a carga entre os ADRs.

7.6.2 Gerenciamento de dados

Implementamos os algoritmos para o armazenamento e recuperação de dados armazenados, incluindo a geração de FFIs, codificação dos dados, cálculo dos identificadores e transferência e armazenamento de dados. O restante das tarefas de gerenciamento de dados ainda estão sendo implementadas: (1) reconstrução de fragmentos no caso de saída de ADRs ou aglomerados da grade, (2) o gerenciamento dos certificados de aluguel (*leases*), (3) o armazenamento local de fragmentos para a otimizações de desempenho e (4) o suporte a adição incremental de dados a arquivos já armazenados (*appending*).

Os processos de armazenamento e recuperação de arquivos são realizados de modo assíncrono, com o *access broker* chamando operações da interface CORBA CdrmRequestManager, implementada pela classe CdrmRequestManagerImpl, e o CDRM enviando as respostas através da interface AccessBroker, implementada pelo *access broker*. Listamos o código destas interfaces abaixo:

```

1 typedef sequence<octet> key; // Identificador de arquivos e fragmentos
2 typedef sequence<key> keySequence;
3 typedef sequence<long> longSequence;
4 typedef sequence<string> stringSequence;
5
6 interface CdrmRequestManager {
7     long requestFileStorage(in key fileKey, in keySequence fragmentKeyList,
8                             in long fileSize, in longSequence fragmentSizeList,
9                             in string accessBrokerIor, in boolean usePereneMode);
10    void setFragmentStorageFinished( in long requestNumber,
11                                     in longSequence storedFragmentList);
12    long requestFileRetrieval(in key fileKey, in string accessBrokerIor);
13 };
14
15 interface AccessBroker {
16    void uploadFragments ( in long requestId, in stringSequence adrAddresses );
17    void setFileStorageRequestCompleted ( in long requestId );
18    void downloadFragments ( in long requestId, in stringSequence adrAddresses,
19                             in keySequence fragmentKeyList,
20                             in long fileSize, in longSequence fragmentSizeList );
21    void setFileRetrievalRequestFailed ( in long requestId );
22 };

```

A operação `requestFileStorage` permite que um *access broker* solicite o armazenamento de um arquivo no OppStore. O *access broker* envia como parâmetros os identificadores do arquivo e dos fragmentos, onde os identificadores são enviados como seqüências de octetos, o tamanho do arquivo e dos fragmentos, o que permite que os CDRMs escolham ADRs que contenham espaço em disco suficiente para os fragmentos, o seu IOR (*Interoperable Object Reference*), que é uma referência CORBA para o *access broker*, permitindo que outros objetos CORBA o contatem remotamente, e se o armazenamento deve ser feito no modo *perene* ou *efêmero*. Ao receber a requisição de armazenamento, para obter os endereços de ADRs para o armazenamento dos fragmentos, a classe `CdrmRequestManagerImpl` cria uma mensagem do tipo `StoreFragmentMessage` para cada fragmento e a repassa para a classe `VirtualNode`, para que esta realize seu roteamento para o CDRM responsável pelo identificador do fragmento. Finalmente, `requestFileStorage` gera um identificador único para esta requisição, armazena as informações recebidas como parâmetros numa tabela e devolve o identificador único gerado para o *access broker* que realizou a chamada do método.

Após receber as respostas para as mensagens solicitando os endereços dos ADRs para o armazenamento de todos os fragmentos, o CDRM chama a operação `uploadFragments` do *access broker*, passando como parâmetro esta lista de endereços. O *access broker* realiza então a transferência dos fragmentos para os ADRs, que é realizada em paralelo para melhorar o desempenho. Após finalizar as transferências, o *access broker* chama a operação `setFragmentStorageFinished` do CDRM, enviando a lista dos fragmentos armazenados com sucesso. O CDRM cria então o FFI do arquivo, contendo a lista de fragmentos armazenados, e o repassa, contido em uma mensagem do tipo `FileFragmentIndexMessage`, para roteamento na rede *peer-to-peer*, para que o FFI seja armazenado pelo CDRM responsável pelo seu identificador. Após receber a confirmação de armazenamento do fragmento, o CDRM então chama a operação `setFileStorageRequestCompleted` de `AccessBroker`, de modo a notificar que a operação de armazenamento foi finalizada com sucesso.

O processo de recuperação de um arquivo é realizado com o *access broker* chamando a operação `requestFileRetrieval` de `CdrmRequestManager`, que recebe como parâmetros o identificador do arquivo e o endereço IOR do *access broker*. Para obter o FFI do arquivo, o CDRM cria uma mensagem do tipo `FileFragmentIndexMessage` e a repassa para ser roteada ao CDRM responsável pelo armazenamento do FFI. O método `requestFileRetrieval` então gera um identificador único para a requisição, armazena os parâmetros da requisição numa tabela e devolve o identificador único gerado.

Caso consiga obter o FFI relativo ao arquivo solicitado, o CDRM chama `downloadFragments` da interface `AccessBroker`, passando como parâmetros o identificador da requisição e os dados contidos no FFI, como a lista de endereços de ADRs, a lista de identificadores dos fragmentos e a lista de tamanhos dos fragmentos. O *access broker* realiza então a transferência, em paralelo, dos fragmentos a partir dos endereços de ADRs fornecidos. Caso o CDRM não consiga obter o FFI, ele chama a operação `setFileRetrievalRequestFailed` da interface `AccessBroker`, notificando que não foi

possível obter informações sobre o arquivo solicitado.

7.6.3 Interface com o InteGrade

O sistema de armazenamento distribuído pode ser utilizado tanto por usuários como por aplicações da grade. Usuários e aplicações possuem requisitos diferentes, de modo que precisam ser tratados de maneira distinta. Ainda não finalizamos a implementação desta interface do OppStore com o InteGrade, descrita na Seção 7.5.2, de modo que nesta seção discutimos o que estamos planejando implementar.

No caso de usuários, a interação com o OppStore se dá pela uma interface gráfica ASCT, que permite que um usuário defina os arquivos de entrada e saída referentes a uma requisição de execução. Na versão atual, os arquivos de entrada são enviados ao LRM que irá executar a aplicação. Após o término da execução, os arquivos de saída são então obtidos diretamente com o mesmo LRM. Iremos alterar o ASCT de modo que os arquivos de entrada sejam transferidos para armazenamento no OppStore assim que a requisição de execução é enviada. Do mesmo modo, após o término da execução, o ASCT passará a obter os dados de saída a partir do OppStore e não diretamente do LRM.

Além disso, alteraremos o LRM para que este obtenha os arquivos de entrada para a execução de processos a partir do OppStore, em lugar do ASCT. Esta obtenção é realizada pelo LRM de modo transparente à aplicação. Do mesmo modo, arquivos de saída serão transferidos para o OppStore após o término da execução da aplicação. Uma preocupação importante é com o possível aumento do tempo para iniciar uma execução e obter seus resultados após o término. Acreditamos que a utilização do mecanismo de *caching*, onde cópias de fragmentos são mantidos no aglomerado onde foram gerados, servirá para reduzir este tempo.

No caso de *checkpoints*, modificamos o mecanismo para obter a lista de repositórios do aglomerado utilizando informações do CDRM do OppStore, em lugar do CRM (*Cluster Repository Manager*). Além disso, a biblioteca de *checkpoint* periodicamente solicita ao *access broker* o armazenamento de parte dos *checkpoints* gerados por uma aplicação.

Outras funcionalidades planejadas

Como trabalho futuro, gostaríamos de disponibilizar uma ferramenta que permita o armazenamento e obtenção de arquivos independentemente de requisições de execução. Isto permitiria que um mesmo arquivo de entrada fosse compartilhado entre diversas execuções e arquivos de saída da execução de uma aplicação pudessem ser utilizados por outras aplicações. Mas para permitir a manipulação de arquivos de modo independente das execuções, precisaríamos também de um módulo para o gerenciamento de metadados de arquivos armazenados, que permitisse, por exemplo, realizar o mapeamento entre usuários, nomes de arquivos e identificadores.

Além disso, seria interessante definirmos uma interface que permita que aplicações também manipulem dados armazenados no OppStore. Aplicações poderiam solicitar dados durante sua execução e

utilizá-los de modo incremental, à medida que fossem disponibilizados. Além disso, dados gerados poderiam também ser armazenados de modo incremental, escondendo assim parte da latência causada pelo armazenamento remoto dos dados.

7.7 Resumo

Neste capítulo, apresentamos o projeto, implementação e implantação do middleware OppStore. Este middleware, que organiza as máquinas da grade em coleções de aglomerados, permite o armazenamento distribuído de dados de modo confiável em repositórios não-dedicados. Utilizamos identificadores virtuais para permitir a escolha de máquinas com maior disponibilidade para o armazenamento dos fragmentos.

No próximo capítulo, apresentaremos os resultados da avaliação experimental e com simulações do middleware OppStore. Verificaremos as melhorias na disponibilidade de dados obtidas com o uso de identificadores virtuais no OppStore e o desempenho do OppStore numa grade oportunista instanciada numa rede de grande área.

Capítulo 8

Avaliação do middleware OppStore

Realizamos a avaliação dos protocolos de identificadores virtuais e do middleware OppStore utilizando duas metodologias: (1) simulações de uma grade oportunista de grande escala e (2) experimentos com uma grade oportunista conectada por uma rede de grande área. O objetivo das simulações é avaliar propriedades dos protocolos de identificadores virtuais e determinar a disponibilidade de dados armazenados em uma grade oportunista de grande escala composta por máquinas compartilhadas. Nos experimentos, medimos o desempenho do armazenamento e recuperação de arquivos em uma grade oportunista composta por aglomerados geograficamente dispersos e interligados pela Internet.

Iniciamos o capítulo apresentando as simulações realizadas e discutindo resultados obtidos. Em seguida, na Seção 8.2, apresentamos os resultados dos experimentos. Finalizamos a seção com um resumo do capítulo.

8.1 Simulações

O objetivo das simulações é avaliar características dos identificadores virtuais e do OppStore quando utilizado em grades oportunistas de grande escala. A grande vantagem de realizar simulações é que, com um único computador, podemos simular grades compostas por centenas de aglomerados e milhares de máquinas compartilhadas. Iniciamos esta seção descrevendo a implementação do ambiente de simulação e seus parâmetros e, em seguida, apresentamos os resultados obtidos.

8.1.1 Ambiente de simulação

Utilizamos como base para a simulação a implementação completa do OppStore, descrita na Seção 7.6, sobre uma versão do FreePastry estendida com identificadores virtuais. A implementação utilizada inclui todos os protocolos de identificadores virtuais e os algoritmos de armazenamento e recuperação de arquivos do OppStore. Mas substituímos a implementação do *access broker* e dos ADRs por

versões simplificadas em Java, que apenas simulam as operações de codificação, transferência e armazenamento de arquivos. Estas operações simuladas são suficientes, uma vez que queremos apenas avaliar a distribuição de fragmentos armazenados e sua disponibilidade. Já os CDRMs foram utilizados sem modificações.

Implementamos o ambiente de simulação de modo a permitir a instanciação de centenas de CDRMs e milhares de ADRs em uma única máquina virtual Java. A comunicação entre CDRMs é feita utilizando o FreePastry, enquanto a comunicação entre *access brokers* e ADRs é feita por Java RMI. Além disso, ao ser inicializado, cada ADR recebe informações sobre o padrão de uso da máquina simulada a que ele pertence.

Para realizar a simulação tivemos que alterar alguns parâmetros da máquina virtual Java, como o tamanho máximo de memória dinâmica que pode ser alocada pela máquina virtual para 2 GB. Além disso, aumentamos o número de arquivos abertos permitidos por usuário no sistema operacional Linux para 64k, já que a simulação requer que sejam abertas milhares de conexões simultaneamente. Finalmente, para evitar o congestionamento da rede *peer-to-peer* durante a busca por repositórios para armazenar fragmentos de um arquivo, limitamos a quantidade de requisições simultâneas realizadas pelos *access brokers*. Sem esta limitação, cada *access broker* geraria milhares de requisições em poucos segundos, o que não é razoável para um cenário típico de utilização do OppStore. As simulações foram executadas em um Athlon64 de 2GHz com 4 GB de memória RAM e sistema operacional Linux 2.6.

Parâmetros utilizados

Realizamos dois tipos de simulação. Na primeira, instanciamos uma grade simulada composta por 300 aglomerados e avaliamos diversas propriedades dos algoritmos de identificadores virtuais, como a distribuição da faixa de identificadores entre os nós e o tamanho e cobertura do conjunto de folhas virtuais dos nós. Na segunda, realizamos as operações de armazenamento e recuperação de arquivos em uma grade simulada composta por 100 aglomerados, de modo a determinar a porcentagem de arquivos que podem ser recuperados com relação ao grau de redundância utilizado em seu armazenamento.

Cada aglomerado recebeu um número aleatório de ADRs, selecionado de um conjunto de 10, 20, 50, 100 e 200 ADRs. Fizemos a suposição de que máquinas de um mesmo aglomerado possuem quantidades similares de espaço livre em disco. Atribuímos a metade dos aglomerados máquinas com espaço livre em disco escolhido aleatoriamente entre 1 e 10 gigabytes e para os demais aglomerados entre 20 e 50 gigabytes.

Para simular os padrões de uso destas máquinas, utilizamos dados de medidas de utilização de máquinas em diferentes ambientes descritos na literatura [20, 43, 92], incluindo universidades e empresas privadas. Baseados nestas medidas, definimos três diferentes padrões de uso, descritos na Tabela 8.1, que foram atribuídos de maneira aleatória aos aglomerados. Deste modo, uma máquina com o padrão 1 de uso tem 60% de chance de estar ociosa em um momento arbitrário durante o dia e 80% durante o

período noturno. Estes dados são utilizados nas simulações de recuperação de arquivos armazenados no OppStore, descritas na Seção 8.1.3.

		padrão 1	padrão 2	padrão 3
média diurna	de tempo ligado	90.0%	60.0%	90.0%
média noturna	de tempo ligado	82.5%	45.0%	82.5%
média diurna	de tempo ocioso	60.0%	25.0%	40.0%
média noturna	de tempo ocioso	80.0%	40.0%	70.0%

Tabela 8.1: Padrões de uso de máquinas compartilhadas.

Utilizando estes parâmetros calculamos a capacidade de cada nó utilizando as equações descritas na Seção 7.2.2. Repetimos estas equações abaixo:

$$Capacidade_{ADR} = disponibilidade\ média \times multiplicador\ do\ disco$$

$$Capacidade_{CDRM} = \sum^{ADRs} Capacidade_{ADR}$$

Simulamos também a distribuição geográfica dos aglomerados utilizando dois casos: (1) aglomerados distribuídos uniformemente por todo o planeta em 24 fuso horários, que denominamos *fuso global*, e (2) aglomerados concentrados em uma região com um único fuso, que denominamos *fuso local*. Estes casos capturam os dois extremos de grades com máquinas distribuídas por todo o mundo e com máquinas concentradas em uma única região. Utilizamos estes casos para avaliar o impacto de correlações no padrão de uso das máquinas nas simulações de armazenamento e recuperação de arquivos no OppStore, descritos na Seção 8.1.3.

8.1.2 Avaliação do uso de identificadores virtuais

Avaliamos diversas propriedades dos algoritmos de identificadores virtuais, como a distribuição da faixa de identificadores entre os nós e o tamanho e cobertura do conjunto de folhas virtuais dos nós. Iniciamos a simulação da grade com um único aglomerado e instanciamos um a um novos aglomerados até formar uma grade com 300 aglomerados. Para cada aglomerado adicionado, medimos diversas propriedades do sistema, de modo a avaliar as propriedades dos algoritmos virtuais para grades de diferentes tamanhos. Repetimos o experimento por 5 vezes, gerando os aglomerados utilizando o modelo descrito na Seção 8.1.1.

Distribuição de identificadores virtuais

Utilizando identificadores virtuais, cada aglomerado da grade fica responsável por uma faixa de identificadores proporcional à sua capacidade. Ao considerarmos todos os aglomerados de uma grade,

Diferença de valores entre extensão da faixa de identificadores alocada a um nó x extensão ideal

Figura 8.1: Tamanho da faixa alocada a cada aglomerado com relação ao tamanho ideal.

suas capacidades e o tamanho do espaço de identificadores, é possível determinar qual seria o tamanho “ideal” da faixa de identificadores alocada a cada aglomerado. Mas cada execução do protocolo de partição do espaço virtual redistribui, entre um subconjunto de tamanho fixo dos nós do sistema, apenas uma parcela do espaço de identificadores. Desejamos verificar se esta partição parcial é suficiente para que os aglomerados fiquem responsáveis por faixas de identificadores aproximadamente proporcionais à sua capacidade.

Nesta simulação, medimos, para cada aglomerado da grade, a diferença entre o tamanho da faixa de identificadores alocada ao nó e o tamanho ideal para esta faixa, utilizando a fórmula:

$$\text{Diferença} = \frac{\text{tamanho ideal} - \text{tamanho alocado}}{\text{tamanho ideal}}$$

Realizamos as medições utilizando tanto os nossos identificadores virtuais quanto os identificadores originais do Pastry de cada nó e apresentamos os resultados na Figura 8.1. Como podemos ver, ao utilizar identificadores virtuais, a diferença no tamanho das faixas é, em média, de 30% e dificilmente acima de 60%. Deste modo, vemos que mesmo realizando alterações nos identificadores de um subconjunto de nós, cada aglomerado fica responsável por um espaço de identificadores de tamanho próximo ao ideal. Já no caso de identificadores Pastry, a diferença média é de 270%, podendo atingir facilmente diferenças de até 800%.

Avaliamos também a distância média entre o identificador virtual e do Pastry de cada aglomerado. Por exemplo, supondo que os identificadores possam assumir valores entre 0 e 1000, um aglomerado

Figura 8.2: Distância entre o identificador virtual e o identificador do Pastry de cada nó.

pode possuir como identificador Pastry o valor 500 e como identificador virtual o valor 700, de modo que a distância entre os identificadores é de 200, ou seja, 20% do tamanho do espaço de identificadores. Como podemos ver na Figura 8.2, em média, a distância entre os identificadores oscilou entre 10% e 20% do tamanho do espaço de identificadores. Esta diferença entre o identificador virtual e do Pastry de cada nó diminui à medida que adicionamos mais aglomerados à grade. Considerando que a entrada de novos aglomerados no sistema altera os identificadores virtuais dos aglomerados vizinhos ao ingressante, enquanto os identificadores Pastry permanecem fixos, deveríamos esperar um aumento na distância média à medida que mais nós são adicionados ao sistema. Por outro lado, o tamanho da faixa de identificadores alocada a cada nó diminui à medida que aumentamos o número de nós no sistema, fazendo com que o identificador virtual fique mais próximo do identificador Pastry. Os resultados das simulações mostram que o segundo efeito é maior que o primeiro. Alterações na distância entre o identificador virtual e o identificador do Pastry não causam efeitos práticos nos protocolos de identificadores virtuais.

Conjunto de folhas virtuais

Avaliamos também o comportamento do conjunto de folhas virtuais com relação ao número de aglomerados no sistema. Medimos o número de nós no conjunto de folhas virtuais e o tamanho da faixa de identificadores coberta pelos nós deste conjunto.

A Figura 8.3 mostra o tamanho médio das faixas de identificadores cobertas pelos nós dos conjuntos de folhas virtuais e do Pastry de cada nó. Como vimos na Seção 6.3, o conjunto de folhas virtuais é construído de modo a cobrir um espaço de identificadores sempre maior que o conjunto de folhas do Pastry. Podemos visualizar esta propriedade na Figura 8.3.

Figura 8.3: Recuperações bem sucedidas de arquivos.

Além disso, a variação no tamanho desta faixa é pequeno, cobrindo entre 2.5% e 4.5% do espaço de identificadores para os conjuntos de folhas virtuais no caso de uma grade com 300 aglomerados. Esta variação é pequena porque o tamanho da faixa coberta pelo conjuntos de folhas virtuais é dependente do tamanho da faixa coberta pelo conjunto de folhas Pastry. Este último conjunto possui tamanho fixo de 8 elementos, de modo que a faixa de identificadores coberta por ambos os conjuntos varia pouco.

No gráfico à esquerda na Figura 8.4 podemos ver a variação do número médio de nós no conjunto de folhas virtuais à medida que aumentamos o número de nós no sistema. O valor médio de nós no conjunto de folhas Pastry permanece constante, com 10 nós, enquanto o conjunto de folhas Pastry, não representado no gráfico, possui sempre 9 nós. Podemos ver ainda que a variação no número de nós no conjunto de folhas virtuais aumenta à medida que adicionamos aglomerados à grade. Para 300 aglomerados, o número de nós no conjunto ficou entre 5 e 15. Para analisar estes resultados, devemos lembrar que o conjunto de folhas virtuais é construído adicionando nós a este conjunto até que ele cubra uma faixa do espaço de identificadores de tamanho igual ou maior ao do conjunto de folhas Pastry. Deste modo, o valor médio de 10 nós no conjunto de folhas virtuais é fácil de entender, uma vez que, em média, um nó a mais é o suficiente para cobrir uma faixa de identificadores maior que o conjunto de folhas Pastry. Já a variação no número de folhas virtuais ocorre porque, além dos nós presentes no conjunto de folhas virtuais não serem necessariamente os mesmos do conjunto de folhas Pastry, a cobertura do conjunto de folhas virtuais é feita considerando os identificadores virtuais destes nós.

No gráfico à direita na Figura 8.4, vemos que a cobertura do conjunto de folhas virtuais de um nó varia entre valores similares a 70% maiores que a cobertura do conjunto de folhas Pastry do mesmo nó. Considerando que em média os conjuntos de folhas Pastry possuem 9 elementos e o conjunto de folhas

Figura 8.4: Número de nós no conjunto de folhas virtuais e razão entre o tamanho da faixa de identificadores coberta pelo nós do conjunto de folhas virtuais com relação aos do conjunto de folhas Pastry.

virtuais 10 elementos, deveríamos esperar que a cobertura média do conjunto de folhas virtuais fosse aproximadamente 10% maior que a do conjunto de folhas Pastry. Mas devemos novamente lembrar que o conjunto de folhas virtuais é construído adicionando, a partir do centro do conjunto, folhas logicamente à esquerda e à direita do nó, até que este conjunto cubra uma faixa do espaço de identificadores de tamanho igual ou maior ao do conjunto de folhas Pastry. Consequentemente, os nós nos extremos do conjunto de folhas virtuais normalmente são nós que cobrem faixas maiores do espaço de identificadores, de modo que a diferença na cobertura entre os conjuntos de folhas virtuais e Pastry é maior do que se os nós extremos fossem escolhidos aleatoriamente.

8.1.3 Recuperação de dados armazenados no OppStore

Simulamos o procedimento de armazenamento para 10 mil arquivos, incluindo o roteamento das mensagens de armazenamento dos fragmentos, para então realizar a recuperação destes arquivos. O objetivo da simulação é avaliar a quantidade de arquivos que conseguimos recuperar considerando que podemos obter apenas fragmentos de máquinas ociosas. Realizaremos a avaliação do desempenho do OppStore durante o armazenamento e recuperação de arquivos na Seção 8.2.

Simulamos um período de um mês utilizando os padrões de uso das máquinas descritos na Tabela 8.1. Para cada arquivo, o *access broker* realiza uma operação de recuperação a cada 2 horas, totalizando 336 tentativas de recuperação para cada um dos 10 mil arquivos. Estas tentativas podem ser simuladas com

Figura 8.5: Recuperações bem sucedidas de arquivos.

uma única requisição, bastando o *access broker* obter com cada ADR armazenando um fragmento do arquivo uma lista com seu estado nestes 336 instantes. Avaliamos os cenários em que os aglomerados são localizados em um único fuso horário e em que são distribuídos uniformemente por todo o planeta.

Realizamos o armazenamento de arquivos codificados em 6 e 24 fragmentos redundantes em dois cenários. No primeiro, a escolha do aglomerado responsável por armazenar cada fragmento é realizada utilizando os identificadores Pastry dos aglomerados, enquanto no segundo a escolha é realizada utilizando os identificadores virtuais dos aglomerados. Para cada recuperação simulada, medimos a quantidade de fragmentos que pudemos recuperar do total de fragmentos armazenados. Consideramos que durante a simulação ocorrem apenas indisponibilidades temporárias de máquinas, não ocorrendo saídas permanentes de máquinas e aglomerados da grade.

Os gráficos da Figura 8.5 mostram as taxas de recuperações bem sucedidas para arquivos codificados em 6 e 24 fragmentos, para aglomerados localizados em um único fuso horário ou espalhados por todo o planeta. Para cada caso, consideramos diversos níveis de redundância, requerendo diferentes números de fragmentos para reconstruir os arquivos.

Como podemos ver, utilizando identificadores virtuais obtivemos taxas de recuperação bem suce-

didadas muito maiores do que quando não os utilizamos. Por exemplo, quando são necessários 8 de 24 fragmentos para reconstruir um arquivo (fator de replicação de 3) e utilizando o cenário de fuso global, apenas 0.1% das requisições falharam quando utilizamos identificadores virtuais, comparado com 1.2% quando não os utilizamos. Quando aumentamos o número de fragmentos necessários para 12 de 24, a taxa de falhas na recuperação de arquivos aumentou para 6.8% e 23.7% respectivamente. Verificamos melhorias nas taxas de recuperação em todos os cenários avaliados. Estas melhorias ocorrem porque o sistema passa a dar preferência a máquinas com maior disponibilidade no momento de armazenar os fragmentos.

Podemos também observar que ao distribuir os aglomerados em regiões com diferentes fuso horários, obtemos uma melhora na taxa de recuperações bem sucedidas, especialmente para maiores níveis de redundância. Isto ocorre porque reduzimos a correlação entre os padrões de uso de máquinas de diferentes aglomerados.

Para concluir, num ambiente onde máquinas permanecem ociosas entre 25% e 80% do tempo, dependendo do aglomerado e do período do dia, conseguimos recuperar um arquivo 99.9% do tempo com um fator de replicação de apenas 3. Isto mostra que, com relação à disponibilidade dos arquivos armazenados e o uso de espaço de armazenamento, o middleware OppStore é bastante eficiente. Na próxima seção realizamos experimentos para verificar o tempo necessário para armazenar e recuperar arquivos utilizando o OppStore.

8.2 Experimentos

Avaliamos o uso do OppStore num ambiente real de uma grade oportunista composta por cinco aglomerados geograficamente dispersos e conectados pela Internet pública. Realizamos o armazenamento e recuperação de arquivos nesta grade com o objetivo de medir a latência destas operações, incluindo o tempo necessário para codificar e decodificar os arquivos e para realizar a transferência dos fragmentos entre *access brokers* e ADRs.

Instanciamos cinco aglomerados, compostos por máquinas compartilhadas, distribuídos em três cidades do Brasil: três aglomerados em São Paulo (sp1, sp2 e sp3), um em Goiânia (go) e um em São Luiz (sl). Goiânia e São Luiz estão distantes 900km e 3000km de São Paulo, respectivamente, e possuem uma latência de resposta (*round-trip time*) de 50ms e 60ms, respectivamente.

Como estamos interessados apenas na transferência de fragmentos entre os aglomerados, instanciamos um único ADR para cada aglomerado. Os aglomerados foram conectados utilizando a Internet pública. O *access broker* foi instanciado em um computador Athlon64 de 2GHz com 4GB de memória RAM e sistema operacional Linux 2.6, localizado no aglomerado sp1, em São Paulo.

Figura 8.6: Armazenamento e recuperação de dados armazenados em máquinas remotas.

8.2.1 Armazenamento de dados

Realizamos o armazenamento de arquivos de diferentes tamanhos entre 100KB e 500MB. Configuramos o *access broker* para codificar o arquivo em 5 fragmentos, sendo 2 suficientes para reconstruir o arquivo. Alteramos o *access broker* para atribuir identificadores aos fragmentos de modo a garantir que cada fragmento de um arquivo fosse armazenado num aglomerado diferente. Esta modificação permite que repliquemos o cenário de uma grade composta por centenas de aglomerados, onde os fragmentos seriam automaticamente alocados para diferentes aglomerados.

O gráfico à esquerda na Figura 8.6 mostra o tempo necessário para finalizar o processo de codificação dos dados (linha pontilhada) e para finalizar o processo de armazenamento (linha contínua), com relação ao tamanho do arquivo. O *access broker* utilizou 12 segundos para codificar um arquivo de 100MB e 60 segundos para um arquivo de 500MB. O tempo necessário para finalizar o armazenamento do arquivo é muito maior, necessitando de 560 segundos para transferir 1.25GB gerados após o *access broker* codificar um arquivo de 500MB. O tempo de armazenamento de um arquivo é limitado pela conexão mais lenta, neste caso a conexão entre São Paulo e Goiânia, com uma taxa de transferência média de 400KB por segundo.

Como vimos na Seção 7.3.3, a transmissão dos dados pode ser realizada de modo assíncrono, quasi-síncrono e síncrono. Para o caso de armazenamento no modo quasi-síncrono, o tempo mais importante é o de codificação dos dados, que é muito menor e não depende da largura de banda das redes. Isto porque, assim que o *access broker* finaliza a codificação dos dados da aplicação, o processo de armazenamento não depende mais destes dados. Deste modo, a aplicação pode continuar sua execução normalmente, simultaneamente à transferência dos fragmentos do *access broker* para os repositórios remotos. Conseqüentemente, neste caso, o processo de armazenamento remoto de um arquivo é bastante eficiente.

8.2.2 Recuperação de dados

Neste experimento, realizamos a recuperação dos arquivos armazenados remotamente no OppStore. Para reconstruir um arquivo, o *access broker* precisa recuperar 2 dos 5 fragmentos armazenados. O tempo necessário para obter cada fragmento pode variar fortemente dependendo dos repositórios escolhidos. Para avaliar estas variações, realizamos experimentos onde o *access broker* obtém os fragmentos dos dois repositórios com as conexões mais rápidas e dos dois com as conexões mais lentas. Estes dois cenários representam os pontos extremos com relação ao tempo para recuperar um arquivo armazenado.

Na Figura 8.6, o gráfico à direita mostra o tempo necessário para recuperar um arquivo utilizando os dois repositórios com as conexões mais rápidas (linha tracejada) e mais lentas (linha contínua), com relação ao tamanho do arquivo. A recuperação do arquivo utilizando os repositórios mais rápidos (taxa de transferência de alguns MBytes por segundo) necessitou de apenas 10 segundos para um arquivo de 100MB e 58 segundos para um de 500MB. A decodificação dos fragmentos não aumenta o tempo de recuperação do arquivo, uma vez que ela é realizada simultaneamente com a transferência dos fragmentos. Mas quando utilizamos os repositórios mais lentos, o tempo necessário para recuperar os dados aumentou bastante, sendo necessários 500s para recuperar um arquivo de 500MB. Mas este caso raramente deve ocorrer, uma vez que o *access broker* pode escolher os repositórios mais rápidos para obter os fragmentos.

Como vimos na Seção 7.3.3, os arquivos recuperados são construídos de forma incremental, de modo que podem ser utilizados antes do término do processo de recuperação. Conseqüentemente, para aplicações que fazem a leitura seqüencial dos dados à medida que estes são necessários, a latência do tempo de transferência dos dados armazenados pode ser parcialmente escondida.

8.3 Resumo

Fizemos a avaliação do OppStore a partir de simulações e experimentos numa rede de grande área. Analisamos diversas características da utilização do identificadores virtuais pelo OppStore, como a distribuição da faixa de identificadores entre os aglomerados de uma grade. Obtivemos ainda ótimos resultados com relação à disponibilidade de arquivos armazenados, especialmente se considerarmos que podemos obter fragmentos apenas das máquinas ociosas. Mas ao realizar a transferência de dados, o desempenho passa a depender da velocidade da rede. Para esconder a latência da transferência de dados, o armazenamento pode ser realizado de modo assíncrono. Já a recuperação pode ser feita de modo incremental, permitindo a utilização de partes do arquivo à medida que estas são disponibilizadas.

Capítulo 9

Conclusões

O principal objetivo para este trabalho foi desenvolver mecanismos que permitam que recursos ociosos de estações de trabalho geograficamente dispersas possam ser efetivamente utilizados para a execução de aplicações paralelas e o armazenamento distribuído de dados.

Na primeira parte do trabalho, desenvolvemos um mecanismo de recuperação por retrocesso baseado em *checkpointing* para aplicações paralelas, de modo a permitir sua execução em ambientes dinâmicos e heterogêneos. Nestes ambientes, máquinas que executam processos de aplicações paralelas podem tornar-se indisponíveis inesperadamente e com uma frequência alta, por exemplo, várias vezes em um mesmo dia. Este mecanismo permite a reinicialização de processos que falharam a partir de um ponto intermediário de execução. A portabilidade dos *checkpoints* é importante, pois permite que uma aplicação que falhou possa ser reiniciada em outra máquina com características diferentes.

O mecanismo que desenvolvemos possui dois componentes centrais: (1) um pré-compilador, que instrumenta o código-fonte de aplicações para periodicamente gerar *checkpoints* portáteis contendo seu estado e (2) a infra-estrutura de monitoramento, armazenamento de *checkpoints* e reinicialização de aplicações.

Analizamos diversas estratégias de armazenamento e comparamos suas sobrecargas sobre o tempo de execução de aplicações paralelas. Para todas as estratégias, a sobrecarga para *checkpoints* globais de 791MB ficou abaixo de 22%, incluindo o tempo necessário para codificar os *checkpoints* em fragmentos redundantes, quando utilizamos intervalos entre *checkpoints* de 60s. Mas esta sobrecarga pode ser facilmente reduzida para valores da ordem de 2% aumentando o intervalo entre *checkpoints* para 5 ou 10 minutos. Estes experimentos indicam que a sobrecarga gerada pelo processo de *checkpointing* é baixa, mesmo para o caso de *checkpoints* portáteis.

Mas a principal contribuição do mecanismo de recuperação por retrocesso é permitir a execução robusta de aplicações paralelas computacionalmente intensivas em máquinas não-dedicadas. O mecanismo fornece suporte a aplicações sequenciais, paralelas do tipo BSP e paralelas do tipo paramétrica. Sem este mecanismo, a grade oportunista ficaria restrita à execução de aplicações sequenciais de curta duração.

Na segunda parte do trabalho desenvolvemos um middleware para o armazenamento distribuído de dados em máquinas compartilhadas de grades oportunistas. Projetamos o middleware para ser escalável, auto-organizável e tolerante a falhas, permitindo sua operação automática no ambiente dinâmico de um grade composta por milhares de nós. Utilizamos o conceito de identificadores virtuais para selecionar para armazenamento os aglomerados cujas máquinas permanecem ociosas por maiores períodos, melhorando assim a disponibilidade dos dados armazenados.

A partir de simulações, mostramos que, utilizando apenas os períodos ociosos de máquinas compartilhadas para armazenar e recuperar dados, podemos obter disponibilidades de dados de 99.9% utilizando um fator 3 de replicação e 93.2% utilizando um fator 2 de replicação. Estes resultados foram obtidos simulando uma grade oportunista em condições realistas, utilizando padrões de uso obtidos em medições de máquinas reais, que mostram que estas máquinas permanecem ociosas entre 25% e 80% do tempo, dependendo do aglomerado e do período do dia. Pudemos, nas simulações, obter estes excelentes índices de disponibilidade de dados em função do uso de identificadores virtuais para realizar a escolha dos locais de armazenamento de fragmentos.

Obtivemos ainda bons resultados nos experimentos de recuperação de dados armazenados em uma grade composta por aglomerados conectados pela Internet. Como estes dados são codificados em fragmentos redundantes, o *broker* pode escolher os repositórios mais rápidos para obter os fragmentos. Se realizarmos *caching* dos dados, podemos obter os dados com desempenho ainda melhor. O armazenamento de dados é mais demorado, uma vez que é preciso transferir todos os fragmentos do arquivo, e o tempo total de armazenamento é limitado pelo repositório mais lento.

Os resultados experimentais indicam que OppStore provê uma plataforma viável e de baixo custo para resolver o problema do armazenamento de dados em grades computacionais oportunistas. OppStore não requer a aquisição de hardware extra e pode ser facilmente implantado sobre uma grade oportunista como o InteGrade.

O InteGrade é disponibilizado como software livre e pode ser obtido a partir da página do InteGrade na Internet (<http://www.integrade.org.br>). Versões atuais do pré-compilador e da biblioteca de *checkpointing* podem ser obtidas na página do sub-projeto de *checkpointing* (<http://www.integrade.org.br/portal/software/checkpointing>). O middleware OppStore pode ser obtido na página do sub-projeto OppStore (<http://www.integrade.org.br/portal/software/OppStore>).

9.1 Trabalhos Futuros

O pré-compilador que instrumenta o código-fonte de aplicações para gerar *checkpoints* portáteis fornece suporte apenas a aplicações C. Seria interessante estendê-lo para trabalhar também com aplicações

C++. Uma dificuldade importante seria tratar das funcionalidades fornecidas pelo uso de orientação a objetos, como no caso de polimorfismo, mas acreditamos que este tratamento seja possível com os mecanismos de reflexão do OpenC++.

Poderíamos ainda aprimorar o mecanismo de *checkpointing*, desenvolvendo algoritmos adaptativos que, utilizando informações sobre o padrão de uso das máquinas do aglomerado onde a aplicação é executada, automaticamente determinariam o intervalo entre *checkpoints* e o nível de redundância utilizado em seu armazenamento.

Gostaríamos também de implantar o OppStore em um ambiente de grade real e monitorar seu uso por usuários reais por longos períodos. A partir dos dados coletados, teríamos uma base para realizar melhorias nos mecanismos e políticas utilizadas pelo OppStore. Entre as melhorias poderíamos definir mecanismos de *caching* mais eficientes e políticas para incentivar a disponibilização de recursos por parte de usuários. Outro ponto de interesse é avaliar outros tipos de métricas que poderiam ser utilizadas para determinar os locais de armazenamento de fragmentos, como largura de banda.

No caso de identificadores virtuais, gostaríamos de verificar sua aplicabilidade para outras classes de aplicações, fora do domínio de grades computacionais. Para sistemas de arquivos distribuídos *peer-to-peer* [35, 108], os identificadores virtuais poderiam ser aplicados de maneira similar ao OppStore. Outra aplicação seria no caso de *multicast* no nível da aplicação [26], onde os nós da rede *peer-to-peer* são utilizados para disseminar conteúdos digitais. Os identificadores virtuais poderiam ser utilizados na seleção dos nós utilizados para a disseminação destes conteúdos.

9.2 Contribuições do Trabalho

Com este trabalho, esperamos contribuir de duas maneiras importantes para a viabilização das grades oportunistas: permitindo (1) a execução robusta de aplicações paralelas na presença de falhas nos nós de processamento e (2) a utilização do espaço livre em disco de máquinas compartilhadas para o armazenamento de dados de aplicações da grade.

9.2.1 Contribuições tecnológicas

Durante o período de doutoramento, produzimos as seguintes contribuições técnicas:

- *Mecanismo de checkpointing portátil para aplicações C seqüenciais e paralelas baseadas no modelo BSP*: disponibilizamos tanto a biblioteca de *checkpointing* quanto o pré-compilador, que permitem gerar *checkpoints* portáteis contendo o estado destas aplicações. No caso de aplicações C seqüenciais, a biblioteca de *checkpointing* e o pré-compilador podem ser utilizados independentemente do InteGrade;

- *Suporte a recuperação por retrocesso baseada em checkpointing no InteGrade*: permite a execução robusta de aplicações seqüenciais e paralelas na presença de falhas ou indisponibilidades nos nós de execução. Também permite a migração de processos no InteGrade, permitindo o desenvolvimento de algoritmos de escalonamento preemptivos;
- *Implementação de identificadores virtuais sobre o FreePastry*: estendemos o FreePastry de modo a permitir que aplicações *peer-to-peer* sejam implementadas utilizando identificadores virtuais. Utilizamos esta extensão para implementar o OppStore.
- *Implementação do middleware OppStore*: realizamos a implementação do OppStore de modo que este possa ser adaptado a diversos middlewares de grades oportunista. Fizemos a implantação do OppStore no InteGrade, para permitir o armazenamento confiável dos dados de aplicações.

9.2.2 Contribuições científicas

Durante o período de doutoramento, produzimos as seguintes contribuições científicas:

- *Uso de reflexão computacional para instrumentar uma aplicação C para realizar checkpointing portátil*: Apesar de existirem outros trabalhos onde *checkpoints* portáteis de aplicação são gerados, existem diferenças importantes entre nossa abordagem e as demais, como a conversão entre representações de dados somente no momento da recuperação do conteúdo do *checkpoint* e o suporte a aplicações paralelas BSP;
- *Análise de estratégias para o armazenamento de checkpoints de aplicações paralelas*: Comparamos, através de experimentos, diferentes algoritmos para o armazenamento confiável e eficiente de *checkpoints* em máquinas pertencentes ao aglomerado onde a aplicação paralela é executada. Analisamos diversos algoritmos para codificação e partição de dados, comparando a sobrecarga de processamento e de comunicação com o grau de tolerância a falhas oferecido por cada estratégia;
- *Identificadores virtuais*: Propusemos o novo conceito de identificadores virtuais, que permitem realizar o balanceamento dinâmico de carga entre nós heterogêneos utilizando como base a infraestrutura de roteamento do Pastry. Isto permite que os identificadores virtuais produzam uma baixa sobrecarga e sejam simples de implementar;
- *Middleware OppStore*: Propusemos o uso do espaço livre em disco das máquinas provedoras de recursos de uma grade computacional para o armazenamento distribuído de dados de aplicações da grade. Desenvolvemos então um sistema de armazenamento distribuído escalável e tolerante a variações na disponibilidade das máquinas para grades oportunistas, o que demonstrou a viabilidade de nossa proposta;

- *Análise por simulação do uso de identificadores virtuais no OppStore*: Simulamos o armazenamento e recuperação de dados em uma grade computacional oportunista composta por aglomerados de máquinas heterogêneas. Simulamos o padrão de uso das máquinas da grade utilizando dados reais do uso de máquinas em diversos ambientes, incluindo empresas e universidades. Analisamos a distribuição e disponibilidade dos dados armazenados utilizando identificadores virtuais;
- *Avaliação experimental do OppStore em uma rede de grande área*: Realizamos experimentos com o OppStore numa grade oportunista composta por aglomerados distribuídos geograficamente e interligados pela Internet. Mostramos a viabilidade da utilização da abordagem do OppStore no mundo real.

9.3 Publicações durante o doutorado

Publicamos diversos artigos em workshops, conferências e periódicos, demonstrando resultados obtidos durante as pesquisas pertencentes ao doutorado.

Durante as investigações iniciais sobre grades computacionais escrevemos um padrão arquitetural que englobasse os diversos sistemas de middleware de grade. Este padrão foi aceito no PLoP'2004 (*The 11th Conference on Pattern Languages of Programs*), a principal conferência internacional sobre padrões. A apresentação de um padrão em um PLoP é feito de modo diferente das demais conferências. No PLoP, formam-se pequenos grupos de discussão que sentam numa mesa redonda e discutem os pontos fortes e fracos de cada trabalho apresentado. Esta discussão permitiu que eu obtivesse valiosas sugestões para a melhoria do padrão. Aprimoramos então o padrão e o enviamos para uma chamada de trabalhos para publicação no livro *Pattern Languages of Program Design 5*.

- *Grid: An Architectural Pattern*. Raphael Y. de Camargo, Andrei Goldchleger, Marcio Carneiro e Fabio Kon. *The 11th Conference on Pattern Languages of Programs (PLoP'2004)*. Monticello, USA, September 2004.
- *The Grid Middleware Architectural Pattern*. Raphael Y. de Camargo, Andrei Goldchleger, Marcio Carneiro e Fabio Kon. *Pattern Languages of Program Design 5*. Editors: Dragos Manolescu, James Noble e Markus Völter. Addison Wesley Publishing Company, 2006, páginas 337-356.

Enquanto desenvolvíamos o mecanismo de *checkpointing* de aplicações paralelas, obtivemos diversos resultados sobre a sobrecarga causada por este mecanismo em aplicações. Escrevemos um artigo que foi publicado no MGC'2004 (*2nd Workshop on Middleware for Grid Computing*) e depois selecionado para ser estendido para publicação no periódico *Concurrency and Computation: Practice and Experience*.

- Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid Middleware. Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon e Alfredo Goldman. ACM/IFIP/USENIX 2nd International Workshop on Middleware for Grid Computing. Toronto, Canada, Outubro 2004.
- Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid Middleware. Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon e Alfredo Goldman. Concurrency and Computation: Practice and Experience. Volume 18, Número 6, páginas 567-579, Maio 2006. John Wiley and Sons Ltd

O próximo passo no trabalho foi tornar o mecanismo de *checkpointing* portátil, deixando-o mais adequado para sistemas de grades computacionais. Obtivemos bons resultados, mostrando que esta portabilidade não tinha um impacto significativo e os publicamos no SBAC-PAD'05 (*17th International Symposium on Computer Architecture and High Performance Computing*), realizado no Rio de Janeiro.

- Portable Checkpointing and Communications for BSP Applications on Dynamic Heterogeneous Grid Environments. Raphael Y. de Camargo, Fabio Kon e Alfredo Goldman. In Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing, páginas 226-233. Rio de Janeiro, Brasil, Outubro 2005.

Finalizada a implementação do mecanismo de *checkpointing*, passamos a trabalhar para solucionar a principal deficiência do trabalho, que era o armazenamento dos *checkpoints* gerados. Nossa abordagem consistia em armazenar os *checkpoints* em outras máquinas do aglomerado onde a aplicação era executada. Avaliamos diversas estratégias para armazenamento dos *checkpoints* e publicamos os resultados no MGC'05 (*3rd ACM/IFIP/USENIX International Workshop on Middleware for Grid Computing*). O trabalho foi selecionado como um dos dois melhores do workshop e fomos convidados a estendê-lo e submetê-lo para o periódico DSONline (*IEEE Distributed Systems Online*).

- Strategies for Storage of Checkpointing Data Using Non-dedicated Repositories on Grid Systems. Raphael Y. de Camargo, Renato Cerqueira e Fabio Kon. 3rd ACM/IFIP/USENIX International Workshop on Middleware for Grid Computing, Grenoble, França, Novembro 2005.
- Strategies for Checkpoint Storage on Opportunistic Grids. Raphael Y. de Camargo, Fabio Kon, e Renato Cerqueira. IEEE Distributed Systems Online, Volume 7, Número 9, Setembro 2006, IEEE Computer Society.

O último passo no doutorado foi o desenvolvimento de um sistema de armazenamento distribuído de dados voltado a grade oportunistas. Desenvolvemos o OppStore e o conceito de identificadores virtuais e realizamos diversas simulações. Enviamos os resultados para o Middleware'06, mas o artigo

foi rejeitado. Uma crítica importante foi relacionada à utilização de simulações e falta de experimentos reais. Publicamos então um artigo no MDS'06 (*Middleware Doctoral Symposium*), onde apresentamos o OppStore. Ao mesmo tempo, continuei com a implementação do OppStore e realizei experimentos para avaliar seu desempenho em grades oportunistas conectadas utilizando a Internet. Publicamos os resultados obtidos no CCGrid'07 (*7th IEEE International Symposium on Cluster Computing and the Grid*).

- Distributed Data Storage for Opportunistic Grids. Raphael Y. de Camargo e Fabio Kon. 3rd International Middleware Doctoral Symposium (MDS'06), Melbourne, Australia, Novembro 2006.
- Design and Implementation of a Middleware for Data Storage in Opportunistic Grids. Raphael Y. de Camargo e Fabio Kon. 7th IEEE International Symposium on Cluster Computing and the Grid, Rio de Janeiro, Brasil, Maio 2007.

Também participei, como co-autor, na elaboração de uma demonstração do middleware InteGrade no SBRC'05 (23o Simpósio Brasileiro de Redes de Computadores) e de um resumo estendido de 5 páginas, que foi aceito como pôster no CCGrid'07 (*7th IEEE International Symposium on Cluster Computing and the Grid*).

- InteGrade: a Tool for Executing Parallel Applications on a Grid for Opportunistic Computing. José Braga Pinheiro Junior, Raphael Y. de Camargo, Andrei Goldchleger e Fabio Kon. 23o Simpósio Brasileiro de Redes de Computadores (Salão de Ferramentas), Fortaleza, Brasil, Maio 2005.
- The InteGrade Project. Elioza Sonoda, Fernando Castor Filho, Raphael Y. de Camargo, Fabio Kon, Siang W. Song, Alfredo Goldman, Marcelo Finger, Renato Cerqueira, Markus Endler, Edson N. Cáceres, Henrique Mongelli, Fabio Costa e Francisco J. S. Silva. 7th IEEE International Symposium on Cluster Computing and the Grid (Poster session), Rio de Janeiro, Brasil, Maio 2007. (Aceito para publicação)

Referências Bibliográficas

- [1] AGBARIA, A., AND FRIEDMAN, R. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *HPDC '99: Proceedings of the the 8th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 1999), IEEE Computer Society.
- [2] ALLCOCK, W. GridFTP protocol specification. Global Grid Forum recommendation (GFD.20), 2003.
- [3] ALON, N., KAPLAN, H., KRIVELEVICH, M., MALKHI, D., AND STERN, J. P. Scalable secure storage when half the system is faulty. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming* (London, UK, 2000), Springer-Verlag, pp. 576–587.
- [4] ALVES, C. E. R., CÁCERES, E. N., DEHNE, F., AND W, S. S. A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison. In *ICCSA '03: The 2003 International Conference on Computational Science and its Applications* (May 2003), Springer-Verlag, pp. 249–258.
- [5] ALVISI, L., ELNOZAHY, E., RAO, S., HUSAIN, S. A., AND MEL, A. D. An analysis of communication-induced checkpointing. Tech. Rep. TR-99-01, Department of Computer Science, University of Texas at Austin, Austin, USA, 1999.
- [6] ALVISI, L., AND MARZULLO, K. Message logging: Pessimistic, optimistic, casual, and optimal. *IEEE Transactions on Software Engineering* 24, 2 (February 1998), 149–159.
- [7] ANDERSON, D. P., COBB, J., KORPELA, E., LEBOSKY, M., AND WERTHIMER, D. SETI@home: an experiment in public-resource computing. *Communications of the ACM* 45, 11 (2002), 56–61.
- [8] ANDROUTSELLIS-THEOTOKIS, S., AND SPINELLIS, D. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys* 36, 4 (2004), 335–371.
- [9] ANTONIU, G., BERTIER, M., CARON, E., DESPREZ, F., BOUG, L., JAN, M., MONNET, S., AND SENS, P. *Future Generation Grids*. Springer Verlag, 2006, ch. GDS: An Architecture Proposal for a Grid Data-Sharing Service, pp. 133–152.

- [10] BALDONI, R., QUAGLIA, F., AND FORNARA, P. An index-based checkpointing algorithm for autonomous distributed systems. *IEEE Transactions on Parallel and Distributed Systems* 10, 2 (February 1999), 181–192.
- [11] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), ACM Press, pp. 164–177.
- [12] BARTHOLOMEW, D. QEMU: A multihost, multitarget emulator. *Linux Journal* 2006, 145 (2006), 3.
- [13] BASNEY, J., LIVNY, M., AND MAZZANTI, P. Harnessing the capacity of computational grids for high energy physics. In *CHEP '00: International Conference on Computing in High Energy and Nuclear Physics*.
- [14] BEGUELIN, A., SELIGMAN, E., AND STEPHAN, P. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing* 43, 2 (1997), 147–155.
- [15] BELL, W. H., CAMERON, D. G., CARVAJAL-SCHIAFFINO, R., MILLAR, A. P., STOCKINGER, K., AND ZINI, F. Evaluation of an economy-based file replication strategy for a data grid. In *CCGRID '03: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2003), IEEE Computer Society, p. 661.
- [16] BERMAN, F., FOX, G., AND HEY, T. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, 2003.
- [17] BHAGWAN, R., TATI, K., CHENG, Y.-C., SAVAGE, S., AND VOELKER, G. M. Total recall: System support for automated availability management. In *NSDI '04: Proceedings of the 1st Symposium on Networked Systems Design and Implementation* (March 2004).
- [18] BLAKE, C., AND RODRIGUES, R. High availability, scalable storage, dynamic peer networks: pick two. In *HOTOS'03: Proceedings of the 9th Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2003), USENIX Association.
- [19] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM* 13, 7 (1970), 422–426.
- [20] BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. *SIGMETRICS Performance Evaluation Review* 28, 1 (2000), 34–43.

- [21] BONORDEN, O., JUULINK, B., VON OTTO, I., AND RIEPING, I. The Paderborn University BSP (PUB) library—design, implementation and performance. In *13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing* (1999), pp. 99–104.
- [22] BRONEVETSKY, G., MARQUES, D., PINGALI, K., AND STODGHILL, P. Automated application-level checkpointing of MPI programs. In *PPoPP '03: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2003), pp. 84–89.
- [23] BUTT, A. R., JOHNSON, T. A., ZHENG, Y., AND HU, Y. C. Kosha: A peer-to-peer enhancement for the network file system. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 2004), IEEE Computer Society, p. 51.
- [24] CAI, M., CHERVENAK, A., AND FRANK, M. A peer-to-peer replica location service based on a distributed hash table. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 2004), IEEE Computer Society, p. 56.
- [25] CASAS, J., CLARK, D., KONURU, R., OTTO, S., PROUTY, R., AND WALPOLE, J. MPVM: A migration transparent version of PVM. Tech. rep., Oregon Graduate Institute School of Science & Engineering, 1995.
- [26] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., NANDI, A., ROWSTRON, A., AND SINGH, A. Splitstream: high-bandwidth multicast in cooperative environments. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), ACM Press, pp. 298–313.
- [27] CASTRO, M., DRUSCHEL, P., KERMARREC, A. M., AND ROWSTRON, A. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications* 20, 8 (October 2002), 1489–1499.
- [28] CHANCHIO, K., AND SUN, X.-H. Data collection and restoration for heterogenous process migration. *Software: Practice and Experience* 32, 9 (2002), 845–871.
- [29] CHANDY, M., AND LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computing Systems* 3, 1 (1985), 63–75.
- [30] CHERVENAK, A., DEELMAN, E., FOSTER, I., GUY, L., HOSCHEK, W., IAMNITCHI, A., KESSELMAN, C., KUNSZT, P., RIPEANU, M., SCHWARTZKOPF, B., STOCKINGER, H., STOCKINGER, K., AND TIERNEY, B. Giggle: a framework for constructing scalable replica location services. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* (Los Alamitos, CA, USA, 2002), IEEE Computer Society Press, pp. 1–17.

- [31] CHERVENAK, A., FOSTER, I., KESSELMAN, C., SALISBURY, C., AND TUECKE, S. The data grid: Towards an architecture for the distributed management and analysis of large scientific data-sets. *Journal of Network and Computer Applications* 23, 3 (2000), 187–200.
- [32] CHIBA, S. A metaobject protocol for C++. In *OOPSLA '95: Proceedings of the 10th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (October 1995), pp. 285–299.
- [33] CIRNE, W., BRASILEIRO, F., ANDRADE, N., COSTA, L., ANDRADE, A., NOVAES, R., AND MOWBRAY, M. Labs of the world, unite!!! *Journal of Grid Computing* 4, 3 (September 2006).
- [34] CONDOR. <http://www.cs.wisc.edu/condor>, 2006.
- [35] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with cfs. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), ACM Press, pp. 202–215.
- [36] DAHLIN, M., CHANDRA, B. B. V., GAO, L., AND NAYATE, A. End-to-end wan service availability. *IEEE/ACM Transactions on Networking* 11, 2 (2003), 300–313.
- [37] DE CAMARGO, R. Y., CERQUEIRA, R., AND KON, F. Strategies for checkpoint storage on opportunistic grids. *IEEE Distributed Systems Online* 18, 6 (September 2006).
- [38] DE CAMARGO, R. Y., GOLDCHLEGER, A., CARNEIRO, M., AND KON, F. The Grid architectural pattern: Leveraging distributed processing capabilities. In *Pattern Languages of Program Design 5* (2006), Addison-Wesley Publishing Company, pp. 337–356.
- [39] DE CAMARGO, R. Y., GOLDCHLEGER, A., KON, F., AND GOLDMAN, A. Checkpointing BSP parallel applications on the InteGrade Grid middleware. *Concurrency and Computation: Practice and Experience* 18, 6 (May 2006), 567–579.
- [40] DE CAMARGO, R. Y., AND KON, F. Distributed data storage for opportunistic grids. In *MDS '06: Proceedings of the 3rd International Middleware Doctoral Symposium* (New York, NY, USA, 2006), ACM Press.
- [41] DE CAMARGO, R. Y., AND KON, F. Design and implementation of a middleware for data storage in opportunistic grids. In *CCGrid '07: Proceedings of the 7th IEEE/ACM International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2007), IEEE Computer Society.
- [42] DE CAMARGO, R. Y., KON, F., AND GOLDMAN, A. Portable checkpointing and communication for BSP applications on dynamic heterogeneous Grid environments. In *SBAC-PAD'05: The 17th*

- International Symposium on Computer Architecture and High Performance Computing* (Rio de Janeiro, Brazil, October 2005), pp. 226–233.
- [43] DOMINGUES, P., MARQUES, P., AND SILVA, L. Resource usage of windows computer laboratories. In *ICPP 2005 Workshops: Int. Conf. on Parallel Processing Workshops* (2005), pp. 469–476.
- [44] EASTLAKE, D., AND JONES, P. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Draft Standard), September 2001.
- [45] ELLARD, D., AND MEGQUIER, J. DISP: Practical, efficient, secure and fault-tolerant distributed data storage. *ACM Transactions on Storage* 1, 1 (2005), 71–94.
- [46] ELNOZAHY, M., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys* 34, 3 (May 2002), 375–408.
- [47] FIOLET, V., OLEJNIK, R., LEFAIT, G., AND TOURSEL, B. Optimal grid exploitation algorithms for data mining. In *ISPD'06: Proceedings of the 5th International Symposium on Parallel and Distributed Computing* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 246–252.
- [48] FOSTER, I., AND KESSELMAN, C. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications* 2, 11 (1997), 115–128.
- [49] FOSTER, I., AND KESSELMAN, C. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.
- [50] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [51] GARAY, J. A., GENNARO, R., JUTLA, C. S., AND RABIN, T. Secure distributed storage and retrieval. In *WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms* (London, UK, 1997), Springer-Verlag, pp. 275–289.
- [52] GARCIA, I. C. *Visões Progressivas de Computações Distribuídas*. Tese de Doutorado, Instituto de Computação, Universidade de Campinas, Brasil, Dezembro 2001.
- [53] GARCIA, I. C., AND BUZATO, L. E. Progressive construction of consistent global checkpoints. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems* (Washington, DC, USA, 1999), IEEE Computer Society, p. 55.
- [54] GODFREY, B., LAKSHMINARAYANAN, K., SURANA, S., KARP, R., AND STOICA, I. Load balancing in dynamic structured P2P systems. In *INFOCOM 2004: Proceedings of the 23th Annual Joint Conference of the IEEE Computer and Communications Societies* (Hong Kong, March 2004), IEEE Computer Society, pp. 2253–2262.

- [55] GOLDCHLEGER, A., KON, F., GOLDMAN, A., FINGER, M., AND BEZERRA, G. C. InteGrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. *Concurrency and Computation: Practice and Experience* 16 (March 2004), 449–459.
- [56] GOLDCHLEGER, A., QUEIROZ, C. A., KON, F., AND GOLDMAN, A. Running highly-coupled parallel applications in a computational grid. In *SBRC '04: Proceedings of the 22th Brazilian Symposium on Computer Networks* (May 2004). Short paper.
- [57] GOLDMAN, A., GOLDCHLEGER, A., KON, F., AND HAYASHIDA, U. The Implementation of the BSP Parallel Computing Model on the InteGrade Grid Middleware. In *ACM/IFIP/USENIX 3rd International Workshop on Middleware for Grid Computing* (Grenoble, France, November 2005).
- [58] GRIMSHAW, A. S., AND WULF, W. A. Legion – A View From 50,000 Feet. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing* (Los Alamitos, California, August 1996), IEEE Computer Society Press.
- [59] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *NSDI '05: Proceedings of the 2nd Symposium on Networked Systems Design and Implementation* (May 2005).
- [60] HAYASHIDA, U. K., OKUDA, K., PANETTA, J., AND SONG, S. W. Generating Parallel Algorithms for Cluster and Grid Computing. In *ICCSA '05: The 2005 International Conference on Computational Science and its Applications* (May 2005), Springer-Verlag, pp. 509–516.
- [61] HILL, J. M. D., DONALDSON, S. R., AND LANFEAR, T. Process migration and fault tolerance of BSPlib programs running on networks of workstations. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing* (London, UK, 1998), Springer-Verlag, pp. 80–91.
- [62] HILL, J. M. D., MCCOLL, B., STEFANESCU, D. C., GOUDREAU, M. W., LANG, K., RAO, S. B., SUEL, T., TSANTILAS, T., AND BISSELING, R. H. BSPlib: The BSP programming library. *Parallel Computing* 24, 14 (1998), 1947–1980.
- [63] IYER, S., ROWSTRON, A., AND DRUSCHEL, P. Squirrel: a decentralized peer-to-peer web cache. In *PODC '02: Proceedings of the 21st Symposium on Principles of Distributed Computing* (New York, NY, USA, 2002), ACM Press, pp. 213–222.
- [64] JAIN, R. K. *The Art of Computer Systems Performance Analysis*. Willey, 1991.
- [65] JIANG, H., AND JIN, S. Exploiting dynamic querying like flooding techniques in unstructured peer-to-peer networks. In *ICNP '05: Proceedings of the 13th IEEE International Conference on Network Protocols* (Los Alamitos, CA, USA, 2005), IEEE Computer Society, pp. 122–131.

- [66] KARABLIEH, F., BAZZI, R. A., AND HICKS, M. Compiler-assisted heterogeneous checkpointing. In *SRDS '01: Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems* (New Orleans, USA, 2001), pp. 56–65.
- [67] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the 29th annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1997), ACM Press, pp. 654–663.
- [68] KARGER, D. R., AND RUHL, M. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA '04: Proceedings of the 6th annual ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2004), ACM Press, pp. 36–43.
- [69] KASBEKAR, M., NARAYANAN, C., AND DAS, C. Using reflection for checkpointing concurrent object oriented programs. In *OOPSLA'98, Workshop on Reflective Programming in C++ and Java* (Vancouver, Canada, October 1998).
- [70] KATZ, R. H., GIBSON, G. A., AND PATTERSON, D. A. Disk system architectures for high performance computing. *Proceedings of the IEEE* 77, 12 (1989), 1842–1858.
- [71] KIRCHER, M., AND JAIN, P. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. John Wiley & Sons, 2004.
- [72] KOHL, J., AND NEUMAN, C. The Kerberos Network Authentication Service (V5). RFC 1510, 1993.
- [73] KOSAR, T., AND LIVNY, M. Stork: Making data placement a first class citizen in the grid. In *ICDCS'04: Proceedings of the 24th IEEE International Conference on Distributed Computing Systems* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 342–349.
- [74] KOVÁCS, J., AND KACSUK, P. A Migration Framework for Executing Parallel Programs in the Grid. In *2nd European Accross Grids Conference* (January 2004).
- [75] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WELLS, C., AND ZHAO, B. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS '00: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2000), ACM Press, pp. 190–201.
- [76] LAMEHAMEDI, H., SZYMANSKI, B., SHENTU, Z., AND DEELMAN, E. Data replication strategies in grid environments. In *ICAPP '02: Proceedings of the 5th International Conference*

- on Algorithms and Architectures for Parallel Processing* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 378–383.
- [77] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7 (1978), 558–565.
- [78] LEE, B.-D., AND WEISSMAN, J. B. Dynamic replica management in the service grid. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 2001), IEEE Computer Society, p. 433.
- [79] LETO, A., CORSO, E., MURRI, R., TERPIN, A., ZOICAS, C., AND COZZINI, S. Egrid project: Experience report on the implementation of a grid infrastructure for the analysis of financial data. In *ITCC '05: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 268–273.
- [80] LI, H. F., RADHAKRISHNAN, T., AND VENKATESH, K. Global state detection in non-fifo networks. In *ICDCS'87: The 7th International Conference on Distributed Computing Systems* (1987), pp. 364–370.
- [81] LITZKOW, M., LIVNY, M., AND MUTKA, M. Condor - A hunter of idle workstations. In *ICDCS '88: Proceedings of the 8th International Conference of Distributed Computing Systems* (June 1988), pp. 104–111.
- [82] LITZKOW, M., TANNENBAUM, T., BASNEY, J., AND LIVNY, M. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Tech. Rep. UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.
- [83] LONG, D., MUIR, A., AND GOLDING, R. A longitudinal survey of internet host reliability. In *SRDS '95: Proceedings of the 14th Symposium on Reliable Distributed Systems* (Washington, DC, USA, 1995), IEEE Computer Society, p. 2.
- [84] LUA, E. K., CROWCROFT, J., PIAS, M., SHARMA, R., AND LIM, S. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials* 7, 2 (2005), 72–39.
- [85] LUBY, M., MITZENMACHER, M., SHOKROLLAH, A., AND SPIELMAN, D. Analysis of low density codes and improved designs using irregular graphs. In *STOC '98: Proceedings of the 30th annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1998), ACM Press, pp. 249–258.

- [86] MALKHI, D., NAOR, M., AND RATAJCZAK, D. Viceroy: a scalable and dynamic emulation of the butterfly. In *PODC '02: Proceedings of the 21th Annual Symposium on Principles of Distributed Computing* (New York, NY, USA, 2002), ACM Press, pp. 183–192.
- [87] MALLUHI, Q. M., AND JOHNSTON, W. E. Coding for high availability of a distributed-parallel storage system. *IEEE Transactions Parallel Distributed Systems* 9, 12 (1998), 1237–1252.
- [88] MANIVANNAN, D., AND SINGHAL, M. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (1999), 703–713.
- [89] MAYMOUNKOV, P., AND MAZIÉRES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, 2002), Springer-Verlag, pp. 53–65.
- [90] MOORE, P. C., JOHNSON, W. R., AND DETRY, R. J. Adapting globus and kerberos for a secure ascii grid. In *SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2001), ACM Press, pp. 21–21.
- [91] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: a read/write peer-to-peer file system. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation* (2002), pp. 31–44.
- [92] MUTKA, M. W., AND LIVNY, M. The available capacity of a privately owned workstation environment. *Performance Evaluation* 12, 4 (1991), 269–284.
- [93] NATRAJAN, A., CROWLEY, M., WILKINS-DIEHR, N., HUMPHREY, M. A., FOX, A. D., GRIMSHAW, A. S., AND CHARLES L. BROOKS, I. Studying protein folding on the grid: experiences using charmm on npaci resources under legion: Research articles. *Concurrency and Computation: Practice and Experience* 16, 4 (2004), 385–397.
- [94] OBJECT MANAGEMENT GROUP. *CORBA v3.0 Specification*, July 2002. OMG Document 02-06-33.
- [95] PINHEIRO, E. Truly-transparent checkpointing of parallel applications, 2002. Working Draft. Available at: http://www.research.rutgers.edu/~edpin/epckpt/paper_html/.
- [96] PLANK, J. S. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Software: Practice & Experience* 27, 9 (1997), 995–1012.
- [97] PLANK, J. S., AND G. KINGSLEY, M. B., AND LI, K. Libckpt: Transparent checkpointing under unix. In *Proceedings of the USENIX Winter 1995 Technical Conference* (1995), pp. 213–323.

- [98] PLANK, J. S., LI, K., AND PUENING, M. A. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems* 9, 10 (1998), 972–986.
- [99] PLAXTON, C. G., RAJARAMAN, R., AND RICHA, A. W. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97: Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 1997), ACM Press, pp. 311–320.
- [100] PRUYNE, J., AND LIVNY, M. Managing checkpoints for parallel programs. In *IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing* (London, UK, 1996), Springer-Verlag, pp. 140–154.
- [101] RABIN, M. O. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM* 36, 2 (1989), 335–348.
- [102] RANGANATHAN, K., IAMNITCHI, A., AND FOSTER, I. Improving data availability through dynamic model-driven replication in large peer-to-peer communities. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2002), IEEE Computer Society, p. 376.
- [103] RAO, A., LAKSHMINARAYANAN, K., SURANA, S., KARP, R., AND STOICA, I. Load balancing in structured p2p systems. In *IPTPS '03: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems* (February 2003), pp. 68–79.
- [104] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2001), ACM Press, pp. 161–172.
- [105] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a DHT. In *USENIX'04: Proceedings of the USENIX Annual Technical Conference* (Boston, MA, USA, 2004), pp. 127–140.
- [106] RIPEANU, M., AND FOSTER, I. A decentralized, adaptive replica location mechanism. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 2002), IEEE Computer Society.
- [107] RODRIGUES, R., AND LISKOV, B. High availability in DHTs: Erasure coding vs. replication. In *IPTPS '05: Revised Selected Papers from the 4th International Workshop on Peer-to-Peer Systems* (London, UK, 2005), Springer-Verlag, pp. 226–239.

- [108] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP '01: Proceedings of the 8th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), ACM Press, pp. 188–201.
- [109] ROWSTRON, A. I. T., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms* (Heidelberg, Germany, 2001), pp. 329–350.
- [110] RUSSELL, D. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering SE-6*, 2 (March 1980), 183–194.
- [111] SAMAR, A., AND STOCKINGER, H. Grid data management pilot (gdmp): A tool for wide area replication in high-energy physics. In *AI '01: Proceedins of the IASTED International Conference on Applied Informatics* (2001).
- [112] SHEN, H., AND XU, C.-Z. Hash-based proximity clustering for load balancing in heterogeneous DHT networks. In *IPDPS '06: Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium* (Washington, DC, USA, April 2006), IEEE Computer Society.
- [113] SMARR, L., AND CATLETT, C. E. Metacomputing. *Communications of the ACM* 35, 6 (1992), 44–52.
- [114] SOBE, P. Stable checkpointing in distributed systems without shared disks. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing* (Washington, DC, USA, 2003), IEEE Computer Society, p. 214.2.
- [115] SOUSA, M. S., AND MELO, A. C. M. A. Packageblast: an adaptive multi-policy grid service for biological sequence comparison. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing* (New York, NY, USA, 2006), ACM Press, pp. 156–160.
- [116] SRINIVASAN, R. XDR: External Data Representation Standard. RFC 1832 (Draft Standard), August 1995.
- [117] STELLNER, G. CoCheck: Checkpointing and process migration for MPI. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium* (Washington, DC, USA, 1996), IEEE Computer Society, pp. 526–531.
- [118] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2001), ACM Press, pp. 149–160.

- [119] STRUMPEN, V., AND RAMKUMAR, B. Portable checkpointing and recovery in heterogeneous environments. Tech. Rep. UI-ECE TR-96.6.1, University of Iowa, June 1996.
- [120] SÍTIO DO PROJETO FREENET. <http://freenetproject.org>, 2006.
- [121] SÍTIO DO PROJETO FREEPASTRY. <http://freepastry.org>, 2007.
- [122] SÍTIO DO PROJETO GNUTELLA. <http://www.gnutella.com>, 2006.
- [123] SÍTIO DO PROJETO INTEGRADE. <http://www.integrate.org.br>, 2007.
- [124] SÍTIO DO PROJETO JACORB. <http://www.jacorb.org>, 2007.
- [125] SÍTIO DO PROJETO JXTA. <http://www.jxta.org/>, 2007.
- [126] SÍTIO DO PROJETO KAZAA MEDIA DESKTOP. <http://www.kazaa.com>, 2006.
- [127] SÍTIO DO PROJETO NAPSTER, 2006.
- [128] SÍTIO DO PROJETO OIL. <http://oil.luaforge.net>, 2007.
- [129] SÍTIO DO PROJETO OPENSSL. <http://www.openssl.org>, 2007.
- [130] THAIN, D., BASNEY, J., SON, S.-C., AND LIVNY, M. The kangaroo approach to data movement on the grid. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing* (Redondo Beach, CA, USA, August 2001), pp. 325–333.
- [131] THAIN, D., TANNENBAUM, T., AND LIVNY, M. Condor and the grid. In *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, Eds. John Wiley & Sons Inc., December 2002.
- [132] TONG, W., DING, J., AND CAI, L. Design and implementation of a grid-enabled BSP. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)* (2003).
- [133] TROMBETTI, G. A., MERELLI, I., AND MILANESI, L. High performance cdna sequence analysis using grid technology. *Journal of Parallel Distributed Computing* 66, 12 (2006), 1482–1488.
- [134] UK, B., TAUFER, M., STRICKER, T., SETTANNI, G., CAVALLI, A., AND CAFLISCH, A. Combining task- and data parallelism to speed up protein folding on a desktop grid platform. In *CC-GRID '03: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2003), IEEE Computer Society, p. 240.

- [135] ULISSES KENDI HAYASHIDA. Desenvolvimento de Algoritmos Paralelos para Aglomerados com a Técnica de Paralelização de Laços. Tese de Mestrado, Instituto de Matemática e Estatística, Universidade de São Paulo, Brasil, Janeiro 2005.
- [136] VALIANT, L. A bridging model for parallel computation. *Communications of the ACM* 33, 8 (1990), 103–111.
- [137] VAZHKUDAI, S. S., MA, X., FREEH, V. W., STRICKLAND, J. W., TAMMINEEDI, N., AND SCOTT, S. L. Freeloader: Scavenging desktop storage resources for scientific data. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2005), IEEE Computer Society, p. 56.
- [138] VENUGOPAL, S., BUYYA, R., AND RAMAMOHANARAO, K. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Computing Surveys* 38, 1 (2006), 3.
- [139] WALTERS, B. VMware virtual platform. *Linux Journal* 1999, 63es (1999), 6.
- [140] WANG, Y. M. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers* 46, 4 (1997), 456–468.
- [141] WEATHERSPOON, H., AND KUBIATOWICZ, J. Erasure coding vs. replication: A quantitative comparison. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, 2002), Springer-Verlag, pp. 328–338.
- [142] WEATHERSPOON, H., WELLS, C., AND KUBIATOWICZ, J. Naming and integrity: Self-verifying data in peer-to-peer systems. In *FuDiCo 2002: Proceedings of the International Workshop on Future Directions of Distributed Systems* (2002).
- [143] WOO, N., CHOI, S., JUNG, H., MOON, J., YEOM, H. Y., PARK, T., AND PARK, H. MPICH-GF: Providing fault tolerance on grid environments. In *CCGrid '03: Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid* (May 2003). Poster session.
- [144] XU, Z., AND BHUYAN, L. Effective load balancing in p2p systems. In *CCGrid '06: Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 81–88.
- [145] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications* 22, 1 (2004), 41–53. Special Issue on Service Overlay Networks.
- [146] ZHONG, H., AND NIEH, J. CRAK: Linux checkpoint/restart as a kernel module. Tech. Rep. Technical Report CUCS-014-01, Columbia University, November 2001.

- [147] ZHU, Y., AND HU, Y. Efficient, proximity-aware load balancing for dht-based p2p systems. *IEEE Transactions on Parallel and Distributed Systems* 16, 4 (2005), 349–361.